Mobile SDK Development Guide

Salesforce Mobile SDK 11.1 (iOS Native, Android Native, React Native, and Hybrid)

Last updated: March 29, 2024

© Copyright 2000–2024 Salesforce, Inc. All rights reserved. Salesforce is a registered trademark of Salesforce, Inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

CONTENTS

CHAPTER 1 Introduction to Mobile Development

In this chapter ...

From impromptu videos to mobile geolocation to online shopping, people everywhere use modern mobile devices to create and consume content. Corporate employees, too, use smart devices to connect with customers, stay in touch with coworkers, and engage the public on social networks.

For enterprise IT departments, the explosion of mobile interaction requires a quick response in software services. Salesforce provides the Salesforce Platform to address this need. This cloud supports the most popular mobile operating systems on various form factors—phone, tablet, wearable. Its technologies let you build custom apps, connect to data from any system, and manage your enterprise from anywhere.

Salesforce Mobile SDK supports the Salesforce Platform. With Mobile SDK, developers can create standalone mobile apps that access the platform through Salesforce APIs.

About Salesforce Mobile Apps

The Salesforce Platform offers two ways to build and deploy enterprise-ready mobile applications.

- The Salesforce mobile app, available on App Store and Google Play Store, is the fastest way for Salesforce administrators and developers to deliver apps for employees. It offers simple point-and-click tools for administrators and the Lightning web development platform for advanced developers.
- Salesforce Mobile SDK gives developers the tools to build custom mobile apps with unique user experiences. Mobile SDK lets you produce standalone custom apps that you distribute through the App Store or Google Play Store. These apps can target employees, customers, or partners. You can choose native or web technologies to build these apps while enjoying the same grade of reliability and security found in the Salesforce mobile app.

Mobile SDK offers the following features:

Enterprise Identity & Security
Mobile SDK implements Salesforce login and authentication flows for you. It includes a complete implementation of Salesforce Connected App Policy, so that all users can access their data securely and easily. It supports SAML and advanced authentication flows so that administrators always have full control over data access.

SmartStore Encrypted Database
Mobile databases are useful for building highly responsive apps that also work in any network condition. SmartStore provides an easy way to store and retrieve data locally while supporting a flexible data model. It uses AES-256 encryption to ensure that your data is always protected.

Mobile Sync
Mobile Sync provides a simple API for synchronizing data between your offline database and the Salesforce cloud. With Mobile Sync, developers can focus on the UI and business logic of their application while leaving the complex synchronization logic to Mobile SDK.

Mobile Services
Mobile SDK supports a wide range of platform mobile services, including push notifications, geolocation, analytics, collaboration tools, and business logic in the cloud. These services can supercharge your mobile application and also reduce development time.

Salesforce Communities
With Salesforce Communities and Mobile SDK, developers can build mobile applications that target their customers and partners. These applications benefit from the same enterprise features and reliability as employee apps.

iOS and Android
Mobile SDK supports native development on the two dominant mobile operating systems.

Hybrid and React Native
For JavaScript developers, Mobile SDK offers Cordova-based hybrid apps and React Native apps. These apps access the Mobile SDK libraries through native containers.

Customize the Salesforce Mobile App, or Create a Custom App?

When you're developing mobile apps for Salesforce org users, you have options. The Salesforce mobile app is the customizable mobile app developed, built, and distributed by Salesforce. Custom apps are standalone iOS or Android apps built from scratch on Salesforce Mobile SDK. Although this guide deals only with Mobile SDK development, here are some differences between the Salesforce mobile app and custom apps.

Customizing the Salesforce Mobile App

- Has a pre-defined, customizable user interface.
- Has full access to Salesforce org data.
- You can create an integrated experience with functionality developed in the Salesforce Platform.
- The Action Bar gives you a way to include your own apps and functionality.
- You can customize the Salesforce mobile app with either point-and-click or programmatic customizations.
- Functionality can be added programmatically through Lightning web components and Visualforce pages.
- Salesforce mobile customizations or apps adhere to the Salesforce mobile navigation. So, for example, a Visualforce page can be called from the navigation menu or from the Action Bar.
- You can use existing Salesforce development experience, both point-and-click and programmatic.
- Included in all Salesforce editions and supported by Salesforce.

Developing Custom Mobile Apps

Custom apps can be free-standing apps built on Salesforce Mobile SDK, or browser apps using plain HTML5 and JavaScript with Ajax. With custom apps, you can:

- Define a custom user experience.
- Access data from Salesforce orgs using REST APIs in native and hybrid local apps, or with Visualforce in hybrid apps using JavaScript Remoting. In HTML apps, do the same using JQueryMobile and Ajax.
- Brand your user interface for customer-facing exposure.
- Create standalone mobile apps with native iOS or Android APIs, or through a hybrid container using JavaScript and HTML, or with React Native (Mobile SDK only).
- Distribute apps through mobile industry channels, such as the App Store or Google Play (Mobile SDK only).
- Configure and control complex offline behavior (Mobile SDK only).
- Use push notifications.
- Design a custom security container using your own OAuth module (Mobile SDK only).
- Other important Mobile SDK considerations:
  - Open-source SDK, downloadable for free through npm installers as well as from GitHub.
  - You develop and compile your apps in an external development environment (Xcode for iOS, Android Studio for Android) instead of a browser window.
  - Development costs depend on your app and your platform.

Mobile SDK integrates Salesforce Platform cloud architecture into mobile apps by providing:

- Implementation of Salesforce Connected App policy.
- Salesforce org login and OAuth credentials management, including persistence and refresh capabilities.
- Secure offline storage with SmartStore.
- Syncing between the Salesforce cloud and SmartStore through Mobile Sync.
- Support for Salesforce Communities.
- Wrappers for Salesforce REST APIs with implicit networking.
- Fast switching between multiple users.
- Cordova-based containers for hybrid apps.

- React Native bridges to Mobile SDK native APIs.

About This Guide

This guide introduces you to Salesforce Mobile SDK and teaches you how to design, develop, and manage mobile applications for the cloud. Topics cover all platforms supported by Mobile SDK, including

- Native iOS
- Native Android
- React Native
- Cordova-based hybrid
- HTML and JavaScript

Intended Audience

This guide is primarily for developers who are already familiar with mobile technology, OAuth2, and REST APIs. It's most easily accessible to developers who have some Salesforce Platform experience. If that doesn't exactly describe you, though, don't worry. We provide resources both for beginners and for more advanced developers. For example, perhaps you're a Salesforce admin who's developing your first mobile app, or an experienced mobile developer who's new to Salesforce technology.

We encourage beginners to start with our Mobile SDK Trailhead modules. Afterwards, you can consult this guide as needed for expanding your knowledge. Here are the Mobile SDK trails:

Develop with Mobile SDK
: Introduces all supported platforms—iOS, Android, hybrid, and React Native—and surveys Mobile SDK offline features.

Modern Mobile Development with iOS
: Augments the iOS beginner trail with Xcode and Swift basics plus more advanced Mobile SDK tutorials.

Shorter, less formal tutorials are scattered throughout this guide.

Note: *Salesforce Mobile SDK Development Guide* is available in PDF and online formats. You can access either version at developer.salesforce.com/docs.

Version

This book is current with Salesforce Mobile SDK 11.1.

Sending Feedback

Questions or comments about this book? Suggestions for topics you'd like to see covered in future versions? You can:

- Post to the Mobile SDK Trailblazer Community
- Post your thoughts on the Salesforce developer discussion forums at developer.salesforce.com/forums
- Use the Feedback button at the bottom of each page in the online documentation (developer.salesforce.com/docs/atlas.en-us.mobile_sdk.meta/mobile_sdk/)

Looking for Mobile SDK support? You can:

- Post to the Mobile SDK Trailblazer Community
- Create a GitHub Issue in one of the Mobile SDK GitHub repositories

- Post in the Salesforce StackExchange site using the < mobile sdk > tag
- Post in our Developer Forums

Note: Mobile SDK is an open-source product and is not supported by Salesforce Support.

CHAPTER 2 Introduction to Salesforce Mobile SDK Development

In this chapter ...

Salesforce Mobile SDK lets you harness the power of Salesforce Platform within stand-alone mobile apps.

Salesforce Platform provides a straightforward and productive platform for Salesforce cloud computing. Developers can use Salesforce Platform to define Salesforce application components—custom objects and fields, workflow rules, Visualforce pages, Apex classes, and triggers. They can then assemble those components into awesome, browser-based desktop apps.

Unlike a desktop app, a Mobile SDK app accesses Salesforce data through a mobile device's native operating system rather than through a browser. To ensure a satisfying and productive mobile user experience, you can configure Mobile SDK apps to move seamlessly between online and offline states. Before you dive into Mobile SDK, take a look at how mobile development works, and learn about essential Salesforce developer resources.

About Native, HTML, and Hybrid Development

Salesforce Mobile SDK gives you options for developing your app. You can choose the options that fit your development skills, device and technology requirements, goals, and schedule.

With Mobile SDK, you can choose the development platform that best fits your business needs and resources. These platforms include:

- Native
- HTML Living Standard
- Hybrid
- React Native

Each platform option supports both iOS and Android. Use the following descriptions to get a sense of which option best suits your requirements.

Native Apps

Native apps are specific to a given mobile platform (iOS or Android). These apps use the development tools and languages that the respective platform supports. For example, you develop iOS apps in Xcode using Swift or Objective-C. Of all app types, native apps look and perform best. They provide the best usability, the best features, and the best overall mobile experience. For example, native is the best choice for delivering the following features:

- **Fast graphics API**—The native platform gives you the fastest graphics. This advantage helps, for example, if you're using large quantities of data and require a fast refresh.
- **Fluid animation**—Related to the fast graphics API is the ability to have fluid animation. This ability is especially important in mobile apps for UI graphics and video streaming.
- **Built-in components**—The camera, contacts, geolocation, and other features native to the device can be seamlessly integrated into mobile apps. Another important built-in component is encrypted storage.
- **Ease of use**—The native platform is what people are accustomed to. When you add that familiarity to the native features they expect, your app becomes that much easier to use.

Native app development requires an integrated development environment (IDE). IDEs provide rich platform-specific tools for building and debugging code, project management, version control, and other necessities. While the required level of experience is higher than for other development scenarios, the levels of control and freedom are also higher. Other benefits of native development can include:

- Proven APIs and frameworks give you confidence that your efforts are worthwhile
- Established components that define standard behaviors like navigation, tabs, and modal dialogs
- Source control that allows you to maintain all your code in one secure place

HTML Apps

HTML apps (also known as HTML5 apps) use standard web technologies—typically the HTML Living Standard, JavaScript, and CSS—to deliver web pages to a mobile browser. Getting started with ubiquitous HTML standards and tools is easier than adopting native compilers or hybrid frameworks.

HTML development creates multi-platform mobile apps that are device agnostic and can run on any modern mobile browser. Responsive web design automatically adapts your content to various screen sizes and resolutions, thus reducing your testing requirements. The HTML "write once, run anywhere" approach minimizes distribution efforts: Push a bug fix or new feature to the production server, and it's instantly available to all users. And because your content is on the web, it's search enabled—a major benefit for many types of apps.

If HTML apps are easy to develop and support and can reach a wide range of devices, what are the drawbacks?

- **No secure offline storage**—HTML browsers support offline databases and caching, but with no standard encryption support. Mobile SDK offers all of these features.
- **Unfriendly security flows**—Trivial security measures can pose complex implementation challenges in mobile web apps. They can also be painful for users. For example, some web apps with authentication require users to reenter their credentials every time the app restarts or returns from a background state.
- **Limited native features**—Support for camera, contacts, and other native features can vary, especially among older mobile browsers.
- **Lack of native look and feel**—HTML can only emulate the native look, and customers can't use familiar compound gestures.

Hybrid Apps

Hybrid apps combine the ease of HTML coding with the power of the native platform. The resulting application can access the device's native capabilities and is distributed through the App Store. You can also create hybrid apps that deliver Lightning web components or Visualforce pages.

Hybrid apps wrap HTML, JavaScript, and CSS code inside a lightweight native container that provides access to native platform features. Generally, hybrid apps provide the best of both HTML and native worlds. They're almost as easy to develop as HTML apps and access native functionality through JavaScript APIs. In addition, hybrid apps can use the Mobile SDK offline features to

- Model, query, search, and edit Salesforce data.
- Securely cache Salesforce data for offline use.
- Synchronize locally cached data with the Salesforce server.

Unlike HTLM apps, hybrid apps are downloaded and installed on mobile devices. Perhaps you're wondering whether hybrid apps store their files on the device or on a server. Great question! The storage location depends on whether the hybrid app is *local* or *remote*.

Local
: You can package HTML and JavaScript code inside the mobile application binary, in a structure similar to a native application. You use REST APIs and Ajax to move data back and forth between the device and the cloud.

Remote
: Alternatively, you can implement the full web application from the server (with optional caching for better performance). Your container app retrieves the full application from the server and displays it in a native web view.

In addition, hybrid remote apps can host Lightning web components. Hybrid is the only Mobile SDK platform that currently supports these popular Salesforce entities.

React Native Apps

React Native apps use the React Native framework from Facebook, Inc., to run JavaScript apps as native code. Rather than following the hybrid paradigm, React Native lets you assemble the native UI building blocks with JavaScript code. This framework provides direct access to native resources and lets you test without recompiling. In performance terms, React Native runs a close second to pure native execution.

React Native apps use specialized scripting languages.

- JavaScript is ES2015.
- CSS is JavaScript code that closely resembles CSS syntax and is typically written inline in your JavaScript app.
- Markup is actually a special flavor of XML named JSX. Unlike HTML, which embeds JavaScript in markup, you embed JSX markup in JavaScript.
- TypeScript support lets you use non-binding static typing in your JavaScript code.

In addition, React Native bridges provide access to standard Mobile SDK features, such as:

- SmartStore
- Mobile Sync
- Salesforce login and authentication
- Salesforce REST API access.

You can even access your own native objects—in Swift, Objective-C, or Java—directly in React Native code.

Mobile Architecture Comparison

The following table shows how the various development scenarios stack up.

	Native, React Native	HTML	Hybrid
Graphics	Native APIs	HTML, Canvas, SVG	HTML, SVG
Performance	Fastest	Fast	Moderately fast
Look and feel	Native	Emulated	Emulated
Distribution	App Store, Google Play	Web	App Store, Google Play
Camera	Yes	Browser dependent	Yes
Notifications (from Salesforce)	Yes	No	Yes
Contacts, calendar	Yes	Browser dependent	Yes
Offline storage	Secure file system	Not secure; shared SQL, Key-Value stores	Secure file system; shared SQL (through Cordova plug-ins)
Geolocation	Yes	Yes	Yes
Swipe	Yes	Yes	Yes
Pinch, spread	Yes	Yes	Yes
Connectivity	Online, offline	Mostly online	Online, offline
Development skills	Swift, Objective-C, Java, Kotlin; JavaScript/TypeScript, markup (React Native only)	HTML, CSS, JavaScript	HTML, CSS, JavaScript

Enough Talk; I'm Ready

If you'd rather read about the details later, there are Quick Start topics in this guide for each native development scenario.

- Hybrid Apps Quick Start
- iOS Native Quick Start
- Android Native Quick Start

CHAPTER 3 What's New in Mobile SDK 11.1

In this chapter ...

Mobile SDK 11.1.0 is a minor release that features modernized support for iOS and Android.

In interim releases, we often deprecate items in native libraries for removal in an upcoming major release. Be sure to check your compiler logs for deprecation warnings so that you can address any changes before they go into effect.

How to Upgrade Your Apps

For information on upgrading Mobile SDK apps, follow the instructions at Migrating from the Previous Release.

General Updates in Mobile SDK 11.1

These changes apply to more than one platform.

External Component Version Updates

Cordova for Android: 12.0.1

Cordova for iOS: 7.0.1

React Native: 0.70.14

What's New in Mobile SDK 11.1 for iOS

iOS 17 Compatibility

We've successfully tested Mobile SDK for compatibility with iOS 17 and XCode 15. See iOS 17 Release Notes.

Swift Package Manager Support

We've introduced support for Swift Package Manager, which can now be used to bring Mobile SDK into applications.

- Binary Frameworks for Mobile SDK are hosted on a new repository: https://github.com/forcedotcom/SalesforceMobileSDK-iOS-SPM.
- We've added a `iOSNativeSwiftPackageManager` template, which pulls its dependencies through Swift Package Manager.
- See also: Add Mobile SDK Libraries to Your Project, Creating an iOS Swift Project Manually, Creating an iOS Project with Forceios.

Deprecated APIs

Check your compiler warnings, or see iOS Current Deprecations.

What's New in Mobile SDK 11.1 for Android

Android 14 Compatibility

We've successfully tested Mobile SDK for compatibility with Android 14. See Android Version 14.

Mobile Sync Library Modernization

We've modernized the Mobile Sync Library on Android.

- All source files are now written in Kotlin.
- Parameters and members now use non-nullable types wherever nulls aren't expected or supported.
- Kotlin syntax is now supported where appropriate. For example: string templates, `?:`, `let`, `also`, `map`, `forEach`, `when`, etc.
- Co-routine wrappers are now available for key methods in SyncManager. See Incremental Syncs with reSync, Handling "Ghost" Records After Sync Down Operations, Using Sync Names.

> **Note:** Although we don't typically require consuming code changes in minor releases such as Mobile SDK 11.1, our modernized Mobile Sync library requires consuming code changes in some cases. For example, constants that were once imported from a class in Java are now imported from a companion object in consuming Kotlin code.

Android Template Updates

- Our Mobile SDK Android templates are now up to date with the Kotlin DSL migration.
- Our templates are now set up to download Mobile SDK artifacts from Maven Central, which results in a friendlier build environment.

Advanced Authentication Enhancements

We've fixed these bugs related to advanced auth.

- If Chrome wasn't found during the advanced auth flow, Android users were presented with an error and couldn't continue with login. Advanced authentication now reinstates the expected behavior of using the default browser if Chrome isn't available at runtime.
- A bug caused some Android users' login flow to reset to the initial screen when the app was backgrounded during MFA. We've fixed this issue and changed LoginActivity's launch mode from `singleInstance` to `singleTop`. Apps that extend LoginActivity now require the same change.

We've added new advanced auth methods that allow you to 1) configure which browser your app selects and 2) view the currently selected custom tab browser. See Configuring Advanced Authentication in Android Apps.

Deprecated APIs

Check your compiler warnings, or see Android Current Deprecations.

What Was New in Recent Releases

Here's an archive of What's New bulletins from recent Mobile SDK releases.

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

Mobile SDK 11.0.1

Mobile SDK 11.0.1 is a minor patch release that features these changes.

- Bug fixes for login and refresh with custom domain and enhanced domain.
- Access token re-hydration in hybrid apps and when using an IDP flow.
- Improved read performance for Key-Value Stores.

Mobile SDK 11.0

Mobile SDK 11.0 is a major release that modernizes several authentication flows. In major releases, we typically remove items that have been deprecated for removal. Read the following information to learn about new features and breaking changes that can affect your app. In every release, be sure to check your compiler logs for deprecation warnings so that you can address these changes before they go into effect.

General Updates

These changes apply to more than one platform.

- The default authentication on iOS and Android now uses Web Server Flow instead of User-Agent Flow. See OAuth 2.0 Web Server Flow on page 405.
- Device system biometric authentication for logins. See Biometric Authentication on page 420.
- Reworked multi-app SSO flows and configurations with identity providers. See Identity Provider Apps on page 442.

External Component Version Updates

SQLCipher: 4.5.4 (iOS and Android)

SQLite: 3.41.2

Cordova for Android: 11.0.0

Cordova for iOS: 6.3.0

Cordova Command Line: 12.0.0

iOS

See also: *General Updates in Mobile SDK 11.0*

Version Updates

Deployment target: 15

Base SDK version: 16

Xcode: 14

Removed APIs

See iOS APIs Removed in Mobile SDK 11.0 on page 566.

Deprecated APIs

Check your compiler warnings, or see iOS Current Deprecations.

Mobile SDK 10.2

Mobile SDK 10.2 is an interim release that features non-breaking API changes and modernized platform support.

iOS

iOS 16 Compatibility

We've successfully tested Mobile SDK for compatibility with iOS 16. See iOS 16 Release Notes.

Version Updates

SQLite: 3.39.2

SQLCipher: 4.5.2

React Native: 0.70.1

ShellJS: 0.8.5 (for command line tools)

TypeScript: 4.8.3 (for React Native)

Deprecated APIs

Check your compiler warnings, or see iOS Current Deprecations.

Android

Android 13 Compatibility

We've successfully tested Mobile SDK for compatibility with Android 13. See Android Version 13.

Version Updates

SQLite: 3.39.2

SQLCipher: 4.5.2

OkHttp: 4.10.0

Cordova-android: 11.0.0

React Native: 0.70.1

ShellJS: 0.8.5 (for command line tools)

TypeScript: 4.8.3 (for React Native)

Android SDK (target API): 33

Deprecated APIs

Check your compiler warnings, or see Android Current Deprecations.

React Native

Version Updates

React Native: 0.70.1

Changes in Mobile SDK 10.1.1

Mobile SDK 10.1.1 is a minor patch release. Mobile SDK 10.1.1 restores use of the **Lock App After** timeout setting from the org's Connected App settings for your mobile app. When set, the mobile app locks after it has been in the background for longer than the timeout period. Locking occurs when the mobile app is activated. Unlocking the app remains the same.

Mobile SDK 10.1

Mobile SDK 10.1.0 is a minor release that includes bug fixes, performance enhancements, feature additions, and updates.

In interim releases, we often deprecate items in native libraries for removal in an upcoming major release. Be sure to check your compiler logs for deprecation warnings so that you can address any changes before they go into effect.

General Updates

These changes apply to more than one platform.

REST API Methods for Briefcase Priming Records (iOS, Android)

REST request factory method and response parser for the Briefcase Priming Salesforce API. See Briefcase Priming Records.

REST API Methods for sObject Collections (iOS, Android, React Native)

REST request factory methods and response parser for the following sObject Collections operations:

- `Create`—See Collection Create.
- `Retrieve`—See Collection Retrieve.
- `Update`—See Collection Update.
- `Upsert`—See Collection Upsert.
- `Delete`—See Collection Delete.

Briefcase Sync Down Target (iOS, Android)

New sync down target for downloading and locally synchronizing records from an org's briefcases. See Using the Briefcase Sync Down Target.

Collection Sync Up Target using sObject Collections (iOS, Android)

New sync up target that uses sObject Collections to improve performance. If you don't specify an implementation class ("androidImpl" or "iOSImpl") in your sync up target configuration, Mobile SDK automatically uses `CollectionSyncUpTarget`. See Using the sObject Collection Sync Up Target.

External Component Version Updates

SQLCipher: 4.5.1 (iOS and Android)

SQLite: 3.37.2

Gradle: 7.2.1

iOS

See also *General Updates in Mobile SDK 10.0*.

Version Updates

SQLCipher: 4.5.1

SQLite: 3.37.2

Deprecated APIs

Check your compiler warnings, or see iOS Current Deprecations.

Android

See also *General Updates in Mobile SDK 10.0*.

MobileSyncExplorerKotlin Template

A new Android app template that demonstrates the full power of Mobile Sync, using Kotlin and Jetpack Compose: https://github.com/forcedotcom/SalesforceMobileSDK-Templates/tree/dev/MobileSyncExplorerKotlinTemplate

Version Updates

SQLCipher: 4.5.4

SQLite: 3.41.2

Gradle: 7.2.1

Deprecated APIs

Check your compiler warnings, or see Android Current Deprecations.

React Native

See also *General Updates in Mobile SDK 10.0*.

Version Updates

React Native: 0.70.14

SmartStore

Version Updates

SQLCipher: 4.5.4 (iOS and Android)

SQLite: 3.41.2

Mobile Sync

See also *General Updates in Mobile SDK 10.0*.

Briefcase Sync Down Target (iOS, Android)

New sync down target for downloading and locally synchronizing records from an org's briefcases.

Collection Sync Up Target using sObject Collections (iOS, Android)

New sync up target that uses sObject Collections to improve performance. If you don't specify an implementation class in your sync up target configuration, Mobile SDK automatically uses `CollectionSyncUpTarget`.

Mobile SDK 10.0

Mobile SDK 10.0.0 is a major trust release. It includes breaking API changes, bug fixes, performance enhancements, minor feature additions, and updates.

In major releases, we remove items in native libraries that were deprecated in interim releases. For your convenience, we've compiled lists of deprecated native APIs.

General Updates

These changes apply to more than one platform.

Binary Storage in Key-Value Stores

Key-value stores in native iOS and Android now support secure binary storage APIs. See Using Key-Value Stores for Secure Data Storage.

External Component Version Updates

React Native: 0.67.1

Cordova for iOS: 6.2.0

Cordova for Android: 10.1.1

Cordova command line: 11.0.0

SQLCipher: 4.5.0 (iOS and Android)

SQLite: 3.36.0

node.js: 12.0 to latest LTS version

iOS

Binary Storage in Key-Value Stores

Key-value stores now support secure binary storage with new Mobile SDK APIs. See Using Key-Value Stores for Secure Data Storage.

Widgets in the MobileSyncExplorerSwift Template App

We've added a Recent Contacts widget to this template.

Version Updates

Deployment target: 14

Base SDK version: 15

Xcode: 13

Removed APIs

See iOS APIs Removed in Mobile SDK 11.0.

Deprecated APIs

Check your compiler warnings, or see iOS Current Deprecations.

Android

Binary Storage in Key-Value Stores

Key-value stores now support secure binary storage using existing Mobile SDK APIs. See Using Key-Value Stores for Secure Data Storage.

Version Updates

Minimum API: Android Nougat (API 24)

Target API: Android 12 (API 32)

Default SDK version for hybrid apps: Android 12 (API 32)

Removed APIs

See Android APIs Removed in Mobile SDK 11.0.

Deprecated APIs

Check your compiler warnings, or see Android Current Deprecations.

React Native

Version Updates

React Native: 0.67.1

Hybrid

Version Updates

Cordova for iOS: 6.2.0

Cordova for Android: 10.1.1

Cordova command line: 11.0.0

SmartStore

WAL for Android

Mobile SDK 10.0 implements write-ahead logging (WAL) in SQLCipher for Android. Although SQLCipher's concurrent read-write support remains blocked on Android, lower-level updates bring measurable improvements to SmartStore performance.

Feature Deprecations

Due to improvements in third-party modules, the external storage feature and the `SoupSpec` class have been deprecated for removal in Mobile SDK 11.0. SmartStore is now fully capable of handling large data sets. See Android Current Deprecations and iOS Current Deprecations.

Version Updates

SQLCipher: 4.5.0 (iOS and Android)

SQLite: 3.36.0

Mobile SDK 9.2.0

Mobile SDK 9.2.0 is an interim release that features non-breaking API changes and modernized iOS support.

In interim releases, we often deprecate items in native libraries for removal in an upcoming major release. Be sure to check your compiler logs for deprecation warnings so that you can address any changes before they go into effect.

These changes apply to more than one platform.

General Updates

Passcode Removal

We've removed app-specific passcodes from iOS and Android apps in favor of mobile operating system security. Mobile SDK still honors an org's passcode requirement but ignores passcode length, passcode timeout, and biometric settings from a connected app. For customers who've already configured a device lock screen or biometric unlock, this upgrade is seamless. For others, the new app lock screen prompts the customer to configure an authentication mode. When the customer reactivates the app from the background, the device passcode, rather than an app-specific passcode, is required. See About Login and Passcodes on page 65 (iOS) and Using Passcodes on page 134 (Android).

External Component Version Updates

Cordova

- iOS: 6.2.0
- Android: 10.1.0

React Native

0.66.0 (iOS and Android)

SQLite

3.34.1

SQLCipher

4.4.3 (iOS and Android)

Mobile SDK 9.1.0

Mobile SDK 9.1.0 is an interim release that features non-breaking API changes and modernized iOS support.

In interim releases, we often deprecate items in native libraries for removal in an upcoming major release. Be sure to check your compiler logs for deprecation warnings so that you can address any changes before they go into effect.

General Updates

These changes apply to more than one platform.

Key-Value Stores (iOS, Android)

- Key-value store version 2 debuts in 9.1. With version 2, you can use key-value store APIs to retrieve all keys from the store.
- The **Inspect Key-Value Store** option of the Dev Support menu now lets you search for all keys that match a given partial or whole key name.
- For details of these features, see Using Key-Value Stores for Secure Data Storage on page 457.

iOS

iPad Support in Sample Apps

- The MobileSyncExplorerSwift template app now supports Catalyst and multiple windows for iPad.
- The RestAPIExplorer sample app now supports Catalyst.

REST API Wrapper Update

- We've added a `batchSize` parameter to `requestForQuery` methods of `SFRestApi` (Objective-C) and `RestClient` (Swift). Use this parameter to specify a preferred number of records to be returned in each fetch. Permissible values range from 200 to 2,000 (default setting). To allow for run-time performance adjustments, Mobile SDK doesn't guarantee that your requested size will be the actual batch size.

Deprecations

- Check your compiler warnings, or see iOS Current Deprecations.

Android

REST API Wrapper Update

- We've added a `batchSize` parameter to the `RestRequest.getRequestForQuery` method. Use this parameter to specify a preferred number of records to be returned in each fetch. Permissible values range from 200 to 2,000 (default setting). To allow for run-time performance adjustments, Mobile SDK doesn't guarantee that your requested size will be the actual batch size.

Deprecations

Check your compiler warnings, or see Android Current Deprecations.

SmartStore

Smart SQL no longer requires index paths for all fields referenced in SELECT or WHERE clauses. See Smart SQL Queries on page 230.

Mobile Sync

SOQL Sync Down Target Enhancement

You can now configure the size for SOQL sync down batches. You can specify any value from 200 to 2,000 (default value). See Using the SOQL Sync Down Target on page 306.

Mobile SDK 9.0

General

These changes apply to more than one platform.

Developer Tools

(iOS, Android) The Dev Support menu now provides a new utility: **Inspect Key-Value Store**. In-App Developer Support

External Component Version Updates

- SQLCipher (iOS, Android): 4.4.2
- SQLite (iOS, Android): 3.33.0
- yarn: 1.22
- Cordova:
  - **iOS:** 6.1.1
  - **Android:** 9.0.0

iOS

iPadOS Support

- Implemented multiple window support for iPadOS. This new feature requires changes to existing apps that intend to run on iPads. See Supporting iPadOS in Mobile SDK Apps on page 102.
- Improved support for landscape mode on iPadOS. (Other than updating to Mobile SDK 9.0, no app changes required.)

Version Updates

- Deployment target: iOS 13
- Base SDK: iOS 14
- Xcode: 12
- CocoaPods: 1.8.0 (no maximum)

Deprecations

- Check your compiler warnings, or see iOS Current Deprecations.
- The Carthage framework manager is no longer officially supported.

Android

Version Updates

Target API: Android 11 (API 30)

Deprecations

Check your compiler warnings, or see Android Current Deprecations.

React Native

TypeScript Now Supported

Mobile SDK's implementation of React Native now supports TypeScript for app development in addition to standard JavaScript. Mobile SDK libraries for React Native now also use types. TypeScript requires you to install the TypeScript compiler. React Native Development

Version Updates

React Native version: 0.63.4

Mobile Sync

Parent-Child Sync Up Adds externalIdField Parameter

The new `externalIdField` parameter for parent-child sync up matches the functionality added for basic sync operations in Mobile SDK 8.0. Syncing Up by External ID on page 294.

CHAPTER 4 Getting Started With Mobile SDK 11.1 for iOS and Android

In this chapter ...

Let's get started creating custom mobile apps! If you haven't done so already, begin by signing up for Salesforce and installing Mobile SDK development tools.

In addition to signing up, you need a connected app definition, regardless of which development options you choose. To install Mobile SDK for Android or iOS (hybrid and native), you use the Mobile SDK npm packages.

Developer Edition or Sandbox Environment?

Salesforce offers a range of environments for developers. The environment that's best for you depends on many factors, including:

- The type of application you're building
- Your audience
- Your company's resources

Development environments are used strictly for developing and testing apps. These environments contain test data that isn't business-critical. Development can be done inside your browser or with the Salesforce Extensions for Visual Studio Code editor.

Types of Developer Environments

A *Developer Edition* environment is a free, fully featured copy of the Enterprise Edition environment, with less storage and users. Developer Edition is a logically separate environment, ideal as your initial development environment. You can sign up for as many Developer Edition orgs as you need. This allows you to build an application designed for any of the Salesforce production environments.

A *Partner Developer Edition* is a licensed version of the free Developer Edition that includes more storage, features, and licenses. Partner Developer Editions are free to enrolled Salesforce partners.

Sandbox is a nearly identical copy of your production environment available to Professional, Enterprise, Performance, and Unlimited Edition customers. The sandbox copy can include data, configurations, or both. You can create multiple sandboxes in your production environments for a variety of purposes without compromising the data and applications in your production environment.

Choosing an Environment

In this book, all exercises assume you're using a Developer Edition org. However, in reality a sandbox environment can also host your development efforts. Here's some information that can help you decide which environment is best for you.

- Developer Edition is ideal if you're a:
  - Partner who intends to build a commercially available Salesforce app by creating a managed package for distribution through AppExchange or Trialforce. Only Developer Edition or Partner Developer Edition environments can create managed packages.
  - Salesforce customer with a Group or Personal Edition, and you don't have access to Sandbox.
  - Developer looking to explore the Salesforce Platform for FREE!
- Partner Developer Edition is ideal if you:
  - Are developing in a team and you require a team environment to manage all the source code. In this case, each developer has a Developer Edition environment and checks code in and out of this team repository environment.
  - Expect more than two developers to log in to develop and test.
  - Require a larger environment that allows more users to run robust tests against larger data sets.
- Sandbox is ideal if you:
  - Are a Salesforce customer with Professional, Enterprise, Performance, Unlimited, or Salesforce Platform Edition, which includes Sandbox.
  - Are developing a Salesforce application specifically for your production environment.
  - Aren't planning to build a Salesforce application to be distributed commercially.
  - Have no intention to list on the AppExchange or distribute through Trialforce.

Development Prerequisites for iOS and Android

We recommend some background knowledge and system setup before you begin building Mobile SDK apps.

It's helpful to have some experience with Salesforce Platform. Familiarity with OAuth, login and passcode flows, and Salesforce connected apps is essential to designing and debugging Mobile SDK apps. See Authentication, Security, and Identity in Mobile Apps.

General Requirements (for All Platforms and Environments)

The following software is required for all Mobile SDK development.

- A Salesforce Developer Edition organization with a connected app.

iOS Native Requirements

- iOS SDK:
  - Deployment target: iOS 15
  - Base SDK: iOS 16
- Xcode version: 14 or later. (We recommend the latest version.)

Android Native Requirements

- Java JDK 11.0.11+9 or later—www.oracle.com/downloads.
- Latest version of Android Studio —developer.android.com/sdk.
- Android SDK, including all APIs in the following range:
  - Minimum API: Android Nougat (API 24)
  - Target API: Android 13 (API 33)
- Android SDK Tools
- Android Virtual Device (AVD)

Hybrid Requirements

- For each mobile platform you support, all native requirements except for forceios and forcedroid npm packages.
- Cordova CLI 12.0.0.
- Forcehybrid npm package, version 11.1.
- Proficiency in HTML5 and JavaScript languages.
- For hybrid remote applications:
  - A Salesforce organization that has Visualforce.
  - A Visualforce start page.

React Native Requirements

- For each mobile platform you support, all native requirements except for forceios and forcedroid npm packages.

- Forcereact npm package, version 11.1.
- Proficiency in JavaScript (ES2015).

Sign Up for Salesforce Platform

To access a wealth of tutorials, blogs, and support forums for all Salesforce developer programs, join the Salesforce Platform.

1. In your browser go to `https://developer.salesforce.com/signup`.
2. Fill in the fields about you and your company.
3. In the `Email Address` field, make sure to use a public address you can easily check from a Web browser.
4. Enter a unique `Username`. Note that this field is also in the *form* of an email address, but it does not have to be the same as your email address, and in fact, it's usually better if they aren't the same. Your username is your login and your identity on `developer.salesforce.com`, and so you're often better served by choosing a username that describes the work you're doing, such as *`develop@workbook.org`*, or that describes you, such as *`firstname@lastname.com`*.
5. Read and then select the checkbox for the `Main Services Agreement`.
6. Enter the Captcha words shown and click **Submit Registration**.
7. In a moment you'll receive an email with a login link. Click the link and change your password.

Creating a Connected App

To enable your mobile app to connect to the Salesforce service, you need to create a connected app. The connected app includes a consumer key, a prerequisite to all development scenarios in this guide.

Create a Connected App

A Salesforce administrator creates connected apps on the Salesforce server. Salesforce connected apps include many settings that are used only by other mobile offerings such as the Salesforce app. The following steps cover the settings that apply to Mobile SDK apps.

To create a connected app:

1. Log into your Salesforce instance.
2. In Setup, enter *`Apps`* in the `Quick Find` box, then select **Apps**.
3. Under Connected Apps, click **New**.
4. Perform steps for Basic Information.
5. Perform steps for API (Enable OAuth Settings).
6. If applicable, perform the optional steps for Mobile App Settings.
7. Click **Save**.

If you plan to support push notifications, see Push Notifications and Mobile SDK for additional connected app settings. You can add these settings later if you don't currently have the necessary information.

After you create your connected app, be sure to copy the consumer key and callback URL for safekeeping. You use these values in Mobile SDK apps for OAuth configuration. To look them up later in Lightning Experience:

1. In Lightning Experience, go to Setup.
2. Navigate to **Apps** > **App Manager**.

3. Select an unmanaged connected app (where App Type equals **Connected**).
4. Click the dropdown button at the right end of the row and select **View**.

Note:

- For basic Mobile SDK apps, the callback URL doesn't have to be a valid URL; it only has to match what the app expects in this field. You can use any custom prefix, such as `sfdc://`.
- To support advanced authentication, the callback URL must be a valid endpoint using your custom URI scheme.
- For IdP use cases, the callback URL must be a valid endpoint using the HTTPS protocol.
- The detail page for your connected app displays a consumer key. It's a good idea to copy this key, as you'll need it later.
- After you create a new connected app, wait a few minutes for the token to propagate before running your app.

Basic Information

Specify basic information about your Mobile SDK app in this section.

1. Enter a connected app name and press Return. The name you enter must be unique among connected apps in your org and may contain spaces.

   Note: Salesforce automatically fills in the API Name field with a version of the name without spaces.

2. Enter a contact email.

Other basic information fields are optional and are not used by Mobile SDK.

API (Enable OAuth Settings)

1. Select **Enable OAuth Settings**.
2. Enter the callback URL (endpoint). Mobile SDK uses this URL to call back to your application during authentication. This value must match the OAuth redirect URI specified in your app's project configuration.
3. For Selected OAuth Scopes, select **Access and manage your data (api)**, **Perform requests on your behalf at any time (refresh_token, offline_access)**, and **Provide access to your data via the Web (web)**.
4. To support Mobile SDK apps that perform authentication through the device's native browser, deselect **Require Secret for Web Server Flow**.

Mobile App Settings

Most settings in this section are not used by Mobile SDK. Here are the exceptions.

1. To support PIN protection, select **PIN Protect**.
2. To support push notifications, select **Push Messaging Enabled**. You can find instructions for this section at Step 2: Creating a Connected App in the *Salesforce Mobile Push Notifications Implementation Guide*.

See Also

Create a Connected App in *Salesforce Help*.

SEE ALSO:

Scope Parameter Values

Installing Mobile SDK

Salesforce Mobile SDK provides two installation paths.

- Download the Mobile SDK open source code from GitHub and set up your own development environment.
- Use a Node Packaged Module (npm) script provided by Salesforce to create your own ready-to-run Mobile SDK projects.

Install Node.js, npm, and Git Command Line

Many mobile developers want to use Mobile SDK as a "black box" and begin creating apps as quickly as possible. For this use case, Mobile SDK offers a variety of npm command line scripts that help you create apps. If you plan to take advantage of these npm tools, you first install Node.js. The Node.js installer automatically installs npm. You also install the latest version of the Git command line.

Mobile SDK 11.1 tools use the following minimum versions:

- npm 3.10
- Git command line (latest version)

Note: Mobile SDK npm tool installation is optional but recommended for native iOS development. The tools are highly recommended for all other platforms.

1. Download the Node.js installer from www.nodejs.org.
2. Run the installer, accepting all prompts that ask for permission to install. This module installs both node.js and npm.
3. Test your installation at a command prompt by running the `npm` command. If you don't see a page of command usage information, revisit Step 2 to find out what's missing.
4. If your command line doesn't recognize the `git` command, go to https://git-scm.com/ to download and install the latest Git package for your system.

   Note: The `git` command line utility is included with Xcode.

Now you're ready to download the Salesforce Mobile SDK npm scripts that create iOS and Android apps.

Mobile SDK npm Packages

Most mobile developers want to use Mobile SDK as a "black box" and begin creating apps as quickly as possible. For this use case Salesforce provides a set of npm packages. Each package installs a command line tool that you can use at a Terminal window or in a Windows command prompt.

Mobile SDK command line tools provide a static snapshot of an SDK release. For iOS, the npm package installs binary modules rather than uncompiled source code. For Android, the npm package installs a snapshot of the SDK source code rather than binaries. You use the npm scripts not only to access Mobile SDK libraries, but also to create projects.

Mobile SDK provides the following command line tools:

forcedroid
: Creates native Android projects in Java or Kotlin.

forceios
: Creates native iOS projects in Swift or Objective-C.

forcehybrid
: Creates a hybrid project based on Cordova with build targets for iOS, Android, or both.

forcereact
Creates a React Native project with build targets for iOS, Android, or both.

Npm packages reside at https://www.npmjs.org. We recommend installing Mobile SDK packages globally using the following command:

- **On Windows:**
  - `npm install -g <npm-package-name>`

    where `npm-package-name` is one of the following:
    - `forcedroid`
    - `forceios`
    - `forcehybrid`
    - `forcereact`
  - To install a package locally, omit `-g`.
- **On Mac OS X:**
  - `sudo npm install -g <npm-package-name>`

    where `npm-package-name` is one of the following:
    - `forcedroid`
    - `forceios`
    - `forcehybrid`
    - `forcereact`
  - If you have read-write permissions to `/usr/local/bin/`, you can omit `sudo`.
  - To install a package locally in a user-owned directory, omit `sudo` and `-g`.
- If npm doesn't exist on your system, install the latest release of Node.js from nodejs.org.
- Npm packages do not support source control, so you can't update your installation dynamically for new releases. Instead, you install each release separately. To upgrade to new versions of the SDK, go to the npmjs.org website and download the new package.

Note: The forceios npm utility is provided as an optional convenience. CocoaPods, node.js, and npm are required for forceios but are not required for Mobile SDK iOS development. To learn how to create Mobile SDK iOS native projects without forceios, see Creating an iOS Swift Project Manually.

iOS Preparation

To create Mobile SDK apps for iOS, you must install the necessary Apple software. If you plan to use forceios, you also install CocoaPods.

In Mobile SDK 4.0 and later, the forceios script uses CocoaPods to import Mobile SDK modules. Apps created with forceios run in a CocoaPod-driven workspace. The CocoaPods utility enhances debugging by making Mobile SDK source code available in your workspace. Also, with CocoaPods, updating to a new Mobile SDK version is painless. You merely update the podfile and then run `pod update` in a terminal window.

Follow these instructions to make sure you're fully prepared for Mobile SDK development on iOS.

- Regardless of the type of iOS app you're developing, make sure that you're up to speed with the iOS native requirements listed at iOS Basic Requirements.
- (Optional) To use forceios:

- To install forceios, see Mobile SDK npm Packages.
- Forceios requires you to install CocoaPods. See *Getting Started* at guides.cocoapods.org.

Note: The forceios npm utility is provided as an optional convenience. CocoaPods, node.js, and npm are required for forceios but are not required for Mobile SDK iOS development. To learn how to create Mobile SDK iOS native projects without forceios, see Creating an iOS Swift Project Manually.

SEE ALSO:

Use CocoaPods with Mobile SDK

Refreshing Mobile SDK Pods

Android Preparation

Before you try to create Mobile SDK apps for Android—native, hybrid, or React Native—install the Android native development environment.

Follow these instructions to make sure you're fully prepared for Mobile SDK development on Android.

- Make sure that you're up to speed with the requirements listed at Native Android Requirements.
- To install forcedroid, forcehybrid, or forcereact, see Mobile SDK npm Packages.

Uninstalling Mobile SDK npm Packages

If you need to uninstall an npm package, use the npm script.

Uninstall Global Installations

For global installations, run the following command from any folder:

- **On Mac OS X:**

  ```
  sudo npm uninstall <package-name> -g
  ```

  Use `sudo` if you lack read-write permissions on the `/usr/local/bin/` directory.
- **On Windows:**

  ```
  npm uninstall <package-name> -g
  ```

where `<package-name>` is replaced by one of the following values:

- forcedroid
- forceios
- forcehybrid
- forcereact

Uninstall Local Installations

For local installations, run the following command from the folder where you installed the package:

- **On Mac OS X or Windows:**

```
npm uninstall <package-name>
```

where `<package-name>` is replaced by one of the following values:

- forcedroid
- forceios
- forcehybrid
- forcereact

Mobile SDK GitHub Repositories

More adventurous developers can delve into the SDK, keep up with the latest changes, and possibly contribute to SDK development through GitHub. Using GitHub allows you to monitor source code in public pre-release development branches. In this scenario, your app includes the SDK source code, which is built along with your app.

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

You don't have to sign up for GitHub to access the Mobile SDK, but it's a good idea to join this social coding community. https://github.com/forcedotcom

You can always find the latest Mobile SDK releases in our public repositories:

- `https://github.com/forcedotcom/SalesforceMobileSDK-iOS`
- `https://github.com/forcedotcom/SalesforceMobileSDK-Android`

Important: To submit pull requests for any Mobile SDK platform, check out the **`dev`** branch as the basis for your changes.

If you're using GitHub only to build source code for the current release, check out the **`master`** branch.

Cloning the Mobile SDK for iOS GitHub Repository (Optional)

Many tools exist for cloning and managing local GitHub repos. The following instructions show three popular ways to clone Mobile SDK plus an important post-clone installation step.

1. Clone the repo. For example, you can use one of the following options.
   - Command line:
     a. In the OS X Terminal app, use the `git` command, which is installed with Xcode:

```
git clone git://github.com/forcedotcom/SalesforceMobileSDK-iOS.git
```

   - Xcode:
     a. Select **Source Control** > **Clone**.
     b. For Save As, accept the default or type a custom directory name.
     c. For Where, choose a convenient parent directory.
     d. Click **Clone**.
   - GitHub Desktop for Mac (https://desktop.github.com/):
     a. Click **Clone a Repository from the Internet**.

   b. Click **URL**.
   c. For Repository URL or GitHub username and repository, type `https://github.com/forcedotcom/SalesforceMobileSDK-iOS`.
   d. Set Local Path to a destination of your choice.
   e. Click **Clone**.

2. In the OS X Terminal app, change to the directory in which you cloned the repo (`SalesforceMobileSDK-iOS` by default).
3. Run the install script from the command line: `./install.sh`

Note: The `install.sh` script checks that you have the correct platform development tools installed. It then syncs external submodules required for Mobile SDK development to your local clone. If your development machine has stored obsolete Mobile SDK project templates for Xcode, the script removes those unnecessary files.

Cloning the Mobile SDK for Android GitHub Repository (Optional)

1. In your browser, navigate to the Mobile SDK for Android GitHub repository: https://github.com/forcedotcom/SalesforceMobileSDK-Android.
2. Clone the repository to your local file system by issuing the following command:

```
git clone git://github.com/forcedotcom/SalesforceMobileSDK-Android.git
```

3. Open a terminal prompt or command window in the directory where you installed the cloned repository.
4. Run `./install.sh` on Mac, or *`cscript install.vbs`* on Windows

Note: The install scripts sync external submodules required for Mobile SDK development to your local clone.After you've run *`cscript install.vbs`* on Windows, *`git status`* returns a list of modified and deleted files. This output is an unfortunate side effect of resolving symbolic links in the repo. Do not clean or otherwise revert these files.

Creating Android Projects with the Cloned GitHub Repository

To create native and hybrid projects with the cloned `SalesforceMobileSDK-Android` repository, follow the instructions in `native/README.md` and `hybrid/README.md` files.

Creating iOS Projects with the Cloned GitHub Repository

To create projects with the cloned `SalesforceMobileSDK-iOS` repository, follow the instructions in the repository's `readme.md` file.

SEE ALSO:

Install Node.js, npm, and Git Command Line

Mobile SDK Sample Apps

Salesforce Mobile SDK includes a wealth of sample applications that demonstrate its major features. You can use the hybrid and native samples as the basis for your own applications, or just study them for reference.

Installing the Sample Apps

In GitHub, sample apps live in the Mobile SDK repository for the target platform. For hybrid samples, you have the option of using the Cordova command line with source code from the `SalesforceMobileSDK-Shared` repository.

Accessing Sample Apps From the GitHub Repositories

For Android:

- Clone or refresh the SalesforceMobileSDK-Android GitHub repo (https://github.com/forcedotcom/SalesforceMobileSDK-Android).
- In the repo root folder, run the install script:
  - **Windows:** *`cscript install.vbs`*
  - **Mac OS X:** *`./install.sh`*
- In Android Studio, import the folder that contains your local `SalesforceMobileSDK-Android` clone.
- Look for sample apps in the `hybrid/HybridSampleApps` and `native/NativeSampleApps` project folders.

Important: On Windows, be sure to run Android Studio as administrator.

For iOS:

- Clone or refresh the SalesforceMobileSDK-iOS GitHub repo (https://github.com/forcedotcom/SalesforceMobileSDK-iOS).
- Run *`./install.sh`* in the repository root folder.
- In Xcode, open the `SalesforceMobileSDK-iOS/SalesforceMobileSDK.xcworkspace` file.
- Look for the sample apps in the `NativeSamples` and `HybridSamples` workspace folders.

Building Hybrid Sample Apps With Cordova

To build hybrid sample apps using the Cordova command line, see Build Hybrid Sample Apps.

Android Sample Apps

Native

Find these samples in the `NativeSampleApps` directory of the SalesforceMobileSDK-Android/native repo.

- **MobileSyncExplorer** demonstrates the power of the native Mobile Sync library on Android. It resides in Mobile SDK for Android under `native/NativeSampleApps/MobileSyncExplorer`.
- **RestExplorer** demonstrates the OAuth and REST API functions of Mobile SDK. It's also useful for investigating REST API actions from a tablet.

Hybrid

Find these samples in the `HybridSampleApps` directory of the SalesforceMobileSDK-Android/hybrid repo.

- **AccountEditor**: Demonstrates how to synchronize offline data using the `mobilesync.js` library.
- **MobileSyncExplorerHybrid**: Demonstrates how to synchronize offline data using the Mobile Sync plugin.
- **NoteSync**: Demonstrates how to use non-REST APIs to retrieve Salesforce Notes.

Template Apps

You can use template apps with `forcedroid create` or `forcedroid createwithtemplate` to spawn Mobile SDK "starter" apps. Find these samples in the github.com/forcedotcom/SalesforceMobileSDK-Templates repo.

Used by forcedroid create

- **AndroidNativeKotlinTemplate**: Standard Android native template, written in Kotlin. Default app type for the `forcedroid create` command. When `forcedroid create` prompts for app type, press `Return` or enter `native_kotlin`.
- **AndroidNativeTemplate**: Standard Android native template, written in Java. When `forcedroid create` prompts for app type, enter `native`.

For use with forcedroid createwithtemplate

- **AndroidIDPTemplate**: Demonstrates how to set up an identity provider in an Android app. When `forcedroid createwithtemplate` prompts for the repo URI, enter `AndroidIDPTemplate`.
- **MobileSyncExplorerKotlinTemplate**: Provides a comprehensive example of Mobile Sync usage. When `forcedroid createwithtemplate` prompts for the repo URI, enter `MobileSyncExplorerKotlinTemplate`.

iOS Sample Apps

Native

Find these samples in the `SampleApps` directory of the SalesforceMobileSDK-iOS/native repo.

- **MobileSyncExplorer** demonstrates the power of the native Mobile Sync library on iOS. It resides in Mobile SDK for iOS under `native/SampleApps/MobileSyncExplorer`.
- **RestAPIExplorer** exercises all native REST API wrappers. It resides in Mobile SDK for iOS under `native/SampleApps/RestAPIExplorer`.

Hybrid

Find these samples in the `hybrid/SampleApps` directory of the SalesforceMobileSDK-iOS-Hybrid repo.

- **AccountEditor**: Demonstrates how to synchronize offline data using the `mobilesync.js` library.
- **MobileSyncExplorer**: Demonstrates how to synchronize offline data using the Mobile Sync plugin.
- **NoteSync**: Demonstrates how to use non-REST APIs to retrieve Salesforce Notes.

Template Apps

You can use template apps with `forceios create` or `forceios createwithtemplate` to spawn Mobile SDK "starter" apps. Find these samples in the github.com/forcedotcom/SalesforceMobileSDK-Templates repo.

Used by forceios create

- **iOSNativeSwiftTemplate**: Standard iOS native template, written in Swift. Default app type for the `forceios create` command. When `forceios create` prompts for app type, press `Return` or enter `native_swift`.
- **iOSNativeTemplate**: Standard iOS native template, written in Objective-C. When `forceios create` prompts for app type, enter `native`.

For use with forceios createwithtemplate

- **iOSIDPTemplate**: Demonstrates how to set up an identity provider in an iOS app. When `forceios createwithtemplate` prompts for the repo URI, enter `iOSIDPTemplate`.

- **iOSNativeSwiftEncryptedNotificationTemplate**: Demonstrates how to set up an iOS app to receive encrypted notifications. When `forceios createwithtemplate` prompts for the repo URI, enter (or copy and paste) *`iOSNativeSwiftEncryptedNotificationTemplate`*.
- **MobileSyncExplorerSwift**: Provides a comprehensive example of Mobile Sync usage. When `forceios createwithtemplate` prompts for the repo URI, enter *`MobileSyncExplorerSwift`*.

Hybrid Sample Apps (Source Only)

Mobile SDK provides only the web app source code for most hybrid sample apps. You can build platform-specific versions of these apps using the Cordova command line. To get the source code, clone the SalesforceMobileSDK-Shared GitHub repository and look in the `samples` folder. To build these hybrid apps for specific mobile platforms, follow the instructions at Build Hybrid Sample Apps.

- **accounteditor**: Uses the Mobile Sync to access Salesforce data.
- **contactexplorer**: Uses Cordova to retrieve local device contacts. It also uses the `force.js` toolkit to implement REST transactions with the Salesforce REST API. The app uses the OAuth2 support in Salesforce SDK to obtain OAuth credentials and then propagates those credentials to `force.js` by sending a javascript event.
- **fileexplorer**: Demonstrates the Files API.
- **mobilesyncexplorer**: Demonstrates using `mobilesync.js`, rather than the Mobile Sync plug-in, for offline synchronization.
- **notesync**: Uses non-REST APIs to retrieve Salesforce Notes.
- **simplesyncreact:**: Demonstrates a React Native app that uses the Mobile Sync plug-in.
- **smartstoreexplorer**: Lets you explore SmartStore APIs.
- **userandgroupsearch**: Lets you search for users in groups.
- **userlist**: Lists users in an organization. This is the simplest hybrid sample app.
- **usersearch**: Lets you search for users in an organization.
- **vfconnector**: Wraps a Visualforce page in a native container. This example assumes that your org has a Visualforce page called `BasicVFTest`. The app first obtains OAuth login credentials using the Salesforce SDK OAuth2 support and then uses those credentials to set appropriate webview cookies for accessing Visualforce pages.

React Native Sample Apps

Template Apps

You can use template apps with *`forcereact create`* or *`forcereact createwithtemplate`* to spawn Mobile SDK "starter" apps. Find these samples in the github.com/forcedotcom/SalesforceMobileSDK-Templates repo.

Used by forcereact create

- **ReactNativeTypeScriptTemplate**: Standard React Native template, written in TypeScript. Default app type for the `forcereact create` command. When `forcereact create` prompts for app type, press *`Return`* or enter *`react_native_typescript`*.
- **ReactNativeTemplate**: React Native template written in ES2015 (plain JavaScript.) When `forcereact create` prompts for app type, enter *`react_native`*.

For use with forcereact createwithtemplate

- **ReactNativeDeferredTemplate**: Demonstrates how to used deferred login in a React Native app. When `forcereact createwithtemplate` prompts for the repo URI, enter *`ReactNativeDeferredTemplate`*.

- **MobileSyncExplorerReactNative**: Provides a comprehensive example of Mobile Sync usage. When `forcereact createwithtemplate` prompts for the repo URl, enter *`MobileSyncExplorerReactNative`*.

CHAPTER 5 Updating Mobile SDK Apps (5.0 and Later)

In this chapter ...

Native and React native apps get an easier path to future Mobile SDK upgrades. Instead of creating an app and porting your app's resources to it, you now update a simple configuration file and then run a script that regenerates your app with the new SDK libraries.

Updating Native and React Native Apps

Each native and React native app directory contains a `package.json` file at its root level. This JSON file contains a "dependencies" object that includes a list of name-value pairs describing Mobile SDK source paths. You can set these values to any local or network path that points to a valid copy of the platform's Mobile SDK. After you've updated this file, perform the update by running:

- `install.js` for Android native, iOS native, and native Swift apps
- `installandroid.js` for React native apps on Android
- `installios.js` for React native apps on iOS

You can find the appropriate file in your app's root folder.

For example, here's the dependencies section of a native Android `package.json` file:

```
"dependencies": {
    "salesforcemobilesdk-android":
"https://github.com/forcedotcom/SalesforceMobileSDK-Android.git"
}
```

This path points to the current release branch of the SalesforceMobileSDK-Android repo.

For iOS, it's the same idea:

```
"dependencies": {
    "salesforcemobilesdk-ios":
"https://github.com/forcedotcom/SalesforceMobileSDK-iOS.git"
}
```

For React native, you can set targets for both iOS and Android, as well as React native versions:

```
"sdkDependencies": {
    "SalesforceMobileSDK-Android":
"https://github.com/forcedotcom/SalesforceMobileSDK-Android.git",
    "SalesforceMobileSDK-iOS":
"https://github.com/forcedotcom/SalesforceMobileSDK-iOS.git"
},
"dependencies": {
    "@react-native-community/masked-view": "^0.1.10",
    "@react-navigation/native": "^6.0.2",
    "@react-navigation/stack": "^6.0.7",
    "react": "17.0.2",
    "react-native-force":
"git+https://github.com/forcedotcom/SalesforceMobileSDK-ReactNative.git",
```

```
    "react-native": "0.65.1",
    "react-native-gesture-handler": "^1.10.3",
    "react-native-safe-area-context": "^3.3.0",
    "react-native-screens": "^3.6.0",
    "create-react-class": "^15.7.0"
},
```

Important: Remember that your React native version must be paired with compatible Mobile SDK versions.

To point to the development branch of any Mobile SDK repo—that is, the branch where the upcoming release is being developed—append "#dev" to the URL. For example:

```
"dependencies": {
    "salesforcemobilesdk-android":
"https://github.com/forcedotcom/SalesforceMobileSDK-Android.git#dev"

    "SalesforceMobileSDK-iOS":
"https://github.com/forcedotcom/SalesforceMobileSDK-iOS.git#dev"
},
"dependencies": {
    "@react-native-community/masked-view": "^0.1.10",
    "@react-navigation/native": "^6.0.2",
    "@react-navigation/stack": "^6.0.7",
    "react": "17.0.2",
    "react-native-force":
"git+https://github.com/forcedotcom/SalesforceMobileSDK-ReactNative.git#dev",

    "react-native": "0.65.1",
    ...
}
```

After you've changed the `package.json` file, don't forget to run the Mobile SDK git installer script as shown in the example.

Example: To upgrade an app to a different version of Mobile SDK for iOS:

1. From your app directory, open `package.json` for editing.
2. In the "sdkDependencies" section, change the value for "SalesforceMobileSDK-iOS" to point a different version of the SalesforceMobileSDK-iOS repo. You can point to the development branch or a different tag of the master branch (5.x or later).
3. From the repo root directory, run the appropriate installer script for your app:
   - For native apps: `install.js`
   - For React Native apps: `installios.js`

The steps for Android are identical except for the iOS labels:

1. From your app directory, open `package.json` for editing.
2. In the "sdkDependencies" section, change the value for "SalesforceMobileSDK-Android" to point a different version of the SalesforceMobileSDK-Android repo. You can point to the development branch or a different tag of the master branch (5.x or later).
3. From the repo root directory, run the appropriate installer script for your app:
   - For native apps: `install.js`

- For React Native apps: `installandroid.js`

Updating Hybrid Apps

For hybrid apps, Mobile SDK libraries are delivered through the Mobile SDK Cordova plug-in. However, with a major release, we recommend that you start with a new template app.

1. Run: *`forcehybrid create`*
2. Create the same type of hybrid project with the same name as your existing project, but in a different folder.
3. When the script finishes, *`cd`* to your new project folder.
4. Add any third-party Cordova plug-ins that your original app used. For example, if your app uses the Cordova status bar plug-in, type:

   ```
   cordova plugin add cordova-plugin-statusbar
   ```

5. Copy your web app resources—JavaScript, HTML5, and CSS files, and so on—from the original project into your new project's `www/` folder. For example, on Mac OS X:

   ```
   cp -RL ~/MyProjects/MyMobileSDK50Project/www/* www/
   ```

6. Run: *`cordova prepare`*

Note: For details on required changes for specific releases, see Migrating from the Previous Release.

Using Maven to Update Mobile SDK Libraries in Android Apps

Beginning in Mobile SDK 9.2.0, native Android libraries are available at Maven Central. To consume a Mobile SDK library, you add a single line to the dependencies section of your app's `build.gradle` file.

To import a library from Maven Central with Gradle, you add a `implementation` statement to the `dependencies` section of your project's `build.gradle` file. To update a library with Gradle, you simply change its version number in the `implementation` statement to the updated version, and then resync your libraries.

The Details

Here's what a typical Gradle `dependencies` section looks like:

```
dependencies {
    classpath 'com.android.tools.build:gradle:7.0.2'
}
```

An `implementation` statement takes the form

```
implementation '<groupID>:<artifactID>:<version>'
```

For Mobile SDK libraries:

- *groupID* is "com.salesforce.mobilesdk"
- *artifactID* is "SalesforceSDK", "SalesforceHybrid", "SmartStore", or "MobileSync"
- *version* is "*x.x.x*" (for example, "9.2.0")

The `implementation` statement imports not only the specified library, but also all its dependencies. As a result, you never have to explicitly compile SalesforceAnalytics, for example, because every other library depends on it. It also means that you can get everything you need with just one statement.

To import Mobile SDK 9.2.0 libraries, add one of the following lines:

- For the SalesforceSDK library:

  ```
  implementation 'com.salesforce.mobilesdk:SalesforceSDK:9.2.0'
  ```

- For the SmartStore library (also imports the SalesforceSDK library):

  ```
  implementation 'com.salesforce.mobilesdk:SmartStore:9.2.0'
  ```

- For the Mobile Sync library (also imports the SalesforceSDK and SmartStore libraries):

  ```
  implementation 'com.salesforce.mobilesdk:MobileSync:9.2.0'
  ```

- For the SalesforceHybrid library (imports the SalesforceSDK, SmartStore, Mobile Sync, and Apache Cordova libraries):

  ```
  implementation 'com.salesforce.mobilesdk:SalesforceHybrid:9.2.0'
  ```

- For the SalesforceReact library (imports the SalesforceSDK, SmartStore, and Mobile Sync libraries):

  ```
  implementation 'com.salesforce.mobilesdk:SalesforceReact:9.2.0'
  ```

Note:

- Mobile SDK enforces a few coding requirements for proper initialization and configuration. To get started, see Android Application Structure.

CHAPTER 6 Native iOS Development

In this chapter ...

Salesforce Mobile SDK delivers libraries and sample Xcode projects for developing native mobile apps on iOS. For writing native iOS code, Mobile SDK supports Swift and Objective-C. Swift is recommended for all new and future development.

When you create a native app using the forceios utility, your project starts as a fully functioning app. This app lets you log in to a Salesforce org, and then it displays live Salesforce data. A newly minted forceios app doesn't do much else, but it shows you that basic SDK features are working as designed.

iOS Native Quick Start

1. Make sure you meet the native iOS requirements.
2. Create a project in the style that you prefer:

   Wizard Style (Recommended)
   To create a project with forceios, CocoaPods, node.js, and npm:

   - Install nodejs (includes npm)—https://nodejs.org
   - Install CocoaPods, latest version—https://www.cocoapods.org.
   - Using npm, install forceios. (npm is automatically installed with nodejs).

     ```
     sudo npm install -g forceios
     ```

   - In a Terminal window, use forceios to create an app.

     ```
     forceios create
     ```

   Semi-Manually
   To add Mobile SDK Swift template files, libraries, and settings to an Xcode template project without using forceios and its third-party dependencies, see Creating an iOS Swift Project Manually, option 1.

   Fully Manually
   To manually recode an Xcode Swift template project as a Mobile SDK project, without using forceios and its third-party dependencies, see Creating an iOS Swift Project Manually, option 2.

Note: For help with setup and installation, check out Set Up Your Mobile SDK Developer Tools in Trailhead.

iOS Basic Requirements

The following software is required for all Mobile SDK development.

- A Salesforce Developer Edition organization with a connected app.

iOS development with Mobile SDK 11.1 also requires the following software.

- iOS SDK:
  - Deployment target: iOS 15
  - Base SDK: iOS 16
- Xcode version: 14 or later. (We recommend the latest version.)

Note: Mobile SDK for iOS supports Cocoa Touch dynamic frameworks.

SEE ALSO:

iOS Preparation
Use CocoaPods with Mobile SDK
Refreshing Mobile SDK Pods

Creating an iOS Project with Forceios

To create an app, use forceios in a terminal window. The forceios utility gives you two ways to create your app.

- Specify the type of application you want, along with basic configuration data.

  or

- Use an existing Mobile SDK app as a template. You still provide the basic configuration data.

You can use forceios in interactive mode with command-line prompts, or in script mode with command-line arguments. To see command usage information, type *`forceios`* without arguments.

Note: Be sure to install CocoaPods before using forceios. See iOS Preparation.

Forceios Application Types

For application type, the *`forceios create`* command accepts either of the following input values:

App Type	Language
Type *`native_swift`* (or press *`RETURN`*)	Swift
Type *`native`*	Objective-C

Using `forceios create` Interactively

To use interactive prompts to create an app, open a Terminal window and type *`forceios create`*. For example:

```
$ forceios create
Enter your application type (native, native_swift): <press RETURN>
Enter your application name: testSwift
Enter the package name for your app (com.mycompany.myapp): com.bestapps.ios
Enter your organization name (Acme, Inc.): BestApps.com
Enter output directory for your app (leave empty for the current directory): testSwift
```

This command creates a native iOS Swift app named "testSwift" in the `testSwift/` subdirectory of your current directory.

Using `forceios create` in Script Mode

In script mode, you can use forceios without interactive prompts. For example, to create a native app written in Swift:

```
$ forceios create --apptype="native_swift" --appname="package-test"
--packagename="com.acme.mobile_apps"
    --organization="Acme Widgets, Inc." --outputdir="PackageTest"
```

Or, to create a native app written in Objective-C:

```
$ forceios create --apptype="native" --appname="package-test"
--packagename="com.acme.mobile_apps"
    --organization="Acme Widgets, Inc." --outputdir="PackageTest"
```

Each of these calls creates a native app named "package-test" and places it in the `PackageTest/` subdirectory of your current directory.

Creating an App from a Template

The *`forceios createWithTemplate`* command is identical to *`forceios create`* except that it asks for a GitHub repo URI instead of an app type. You set this URI to point to any repo directory that contains a Mobile SDK app that can be used as a template. Your template app can be any supported Mobile SDK app type. The script changes the template's identifiers and configuration to match the values you provide for the other parameters.

Before you use `createWithTemplate`, it's helpful to know which templates are available. To find out, type *`forceios listtemplates`*. This command prints a list of templates provided by Mobile SDK. Each listing includes a brief description of the template and its GitHub URI. For example:

```
Available templates:

1) Swift application using MobileSync, SwiftUI and Combine
forceios createwithtemplate --templaterepouri=iOSNativeSwiftTemplate

2) Swift application using MobileSync, SwiftUI and Combine (pulled in using Swift Package
 Manager)
forceios createwithtemplate --templaterepouri=iOSNativeSwiftPackageManagerTemplate

3) Basic Swift application with notification service extension
forceios createwithtemplate --templaterepouri=iOSNativeSwiftEncryptedNotificationTemplate

4) Basic Objective-C application
forceios createwithtemplate --templaterepouri=iOSNativeTemplate

5) Sample Swift Identity Provider application
forceios createwithtemplate --templaterepouri=iOSIDPTemplate

6) Sample Swift application using MobileSync data framework
forceios createwithtemplate --templaterepouri=MobileSyncExplorerSwift
```

Once you've found a template's URI, you can plug it into the forceios command line. Here's command-line usage information for `forceios createWithTemplate`:

```
Usage:
forceios createWithTemplate
    --templaterepouri=<Template repo URI> (e.g., MobileSyncExplorerReactNative)]
    --appname=<Application Name>
    --packagename=<App Package Identifier> (e.g. com.mycompany.myapp)
    --organization=<Organization Name> (Your company's/organization's name)
    --outputdir=<Output directory> (Leave empty for current directory)]
```

For any template in the `SalesforceMobileSDK-Templates` repo, you can drop the path for `templaterepouri`—just the template name will do. For example:

```
forceios createwithtemplate --templaterepouri=iOSNativeSwiftTemplate
```

You can use `forceios createWithTemplate` interactively or in script mode. For example, here's a script mode call that uses a specific tag for the template:

```
forceios createWithTemplate
--templaterepouri=MobileSyncExplorerSwift
--appname=MyMobileSyncExplorer
--packagename=com.mycompany.react
--organization="Acme Software, Inc."
--outputdir=testWithTemplate
```

This call creates a native Swift app with the same source code and resources as the MobileSyncExplorerSwift sample app. Forceios places the new app in the `testWithTemplate/` subdirectory of your current directory. It also changes the app name to "MyMobileSyncExplorer" throughout the project.

Checking the Forceios Version

To find out which version of forceios you've installed, run the following command:

```
forceios version
```

Running Your New forceios Project

Apps created with forceios are ready to run, right out of the box. After forceios finishes, it prints instructions for opening and running the project from the command line.

If you prefer, you can leave the command line and open your new project manually in Xcode.

1. In Xcode, select **File** > **Open**.
2. Navigate to the output folder you specified.
3. Open the workspace file (`<project_name>.xcworkspace`).
4. When Xcode finishes building, click the **Run** button.

.

How the Forceios Script Generates New Apps

- Apps are based on CocoaPods or Swift Package Manager.
- The script downloads templates at runtime from a GitHub repo.
- For the `forceios create` command, the script uses the default templates in the SalesforceMobileSDK-Templates GitHub repo.
- For templates based on CocoaPods, the script uses npm at runtime to download Mobile SDK libraries. The podfile refers to these libraries with `:path => node_modules/...` directives.
- For projects built with Swift Package Manager, the script configures the Xcode project to use the Salesforce Mobile SDK Swift Package published on https://github.com/forcedotcom/SalesfoceMobileSDK-iOS-SPM.

SEE ALSO:

Updating Mobile SDK Apps (5.0 and Later)

Using a Custom Template to Create Apps

Wishing you could use your own—or someone else's—custom app as a template? Good idea! Custom templates promote reuse of code, rapid development, and internal consistency. Beginning in Mobile SDK 5.0, you can use either forceios or forcedroid to create apps with custom templates. To turn a Mobile SDK app into a template, you perform a few steps to prepare the app's repo for Mobile SDK consumption.

About Mobile SDK Templates

Mobile SDK defines a template for each architecture it supports on iOS and Android. These templates are maintained in the github.com/forcedotcom/SalesforceMobileSDK-Templates repo. When a customer runs the forcedroid or forceios *`create`* command, the script copies the appropriate built-in template from the repo and transforms this copy into the new app. Apps created this way are basic Mobile SDK apps with little functionality.

Perhaps you'd like to create your own template, with additional functionality, resources, or branding. You can harness the same Mobile SDK mechanism to turn your own app into a template. You can then tell forcedroid or forceios to use that template instead of its own.

How to Use a Custom Template

In addition to forcedroid and forceios *`create`*, Mobile SDK defines a *`createWithTemplate`* command.When you run forcedroid or forceios *`createWithTemplate`*, you specify a template app repo instead of an app type, followed by the remaining app creation parameters. The template app repo contains a Mobile SDK app that the script recognizes as a template. To create a new Mobile SDK app from this template, the script copies the template app to a new folder and applies your parameter values to the copied code.

The `template.js` File

To accept your unknown app as a template, forceios and forcedroid require you to define a `template.js` configuration file. You save this file in the root of your template app repo. This file tells the script how to perform its standard app refactoring tasks—moving files, replacing text, removing and renaming resources. However, you might have even more extensive changes that you want to apply. In such cases, you can also adapt `template.js` to perform customizations beyond the standard scope. For example, if you insert your app name in classes other than the main entry point class, you can use `template.js` to perform those changes.

A `template.js` file contains two parts: a JavaScript "prepare" function for preparing new apps from the template, and a declaration of exports.

The `template.js` Prepare Funtion

Most of a `template.js` file consists of the "prepare" function. By default, prepare functions use the following signature:

```
function prepare(config, replaceInFiles, moveFile, removeFile)
```

You can rename this function, as long as you remember to specify the updated name in the list of exports. The Mobile SDK script calls the function you export with the following arguments:

- `config`: A dictionary identifying the platform (iOS or Android), app name, package name, organization, and Mobile SDK version.
- `replaceInFiles`: Helper function to replace a string in files.
- `moveFile`: Helper function to move files and directories.
- `removeFile`: Helper function to remove files and directories.

The default prepare function found in Mobile SDK templates replaces strings and moves and removes the files necessary to personalize a standard template app. If you intend to add functionality, place your code within the prepare function. Note, however, that the helper

functions passed to your prepare function can only perform the tasks of a standard template app. For custom tasks, you'll have to implement and call your own methods.

Exports Defined in `template.js`

Each `template.js` file defines the following two exports.

appType
Assign one of the following values:

- `'native'`
- `'native_kotlin'` (forcedroid only)
- `'native_swift'` (forceios only)
- `'react_native'`
- `'hybrid_local'`
- `'hybrid_remote'`

prepare
The handle of your prepare function (listed without quotation marks).

Here's an example of the export section of a `template.js` file. This template is for a native app that defines a prepare function named `prepare`:

```
//
// Exports
//
module.exports = {
    appType: 'native',
    prepare: prepare
};
```

In this case, the prepare function's handle is, in fact, "prepare":

```
function prepare(config, replaceInFiles, moveFile, removeFile)
```

Template App Identification in `template.js` (Native and React Native Apps)

For native and React native apps, a template app's prepare function defines an app name, a package name, and an organization or company name. These values identify the template app itself—not a new custom app created from the template. At runtime, the Mobile SDK script uses these values to find the strings to be replaced with the script's input values. Here's an example of the settings for these `iOSNativeTemplate` template app:

```
// Values in template
var templateAppName = 'iOSNativeTemplate';
var templatePackageName = 'com.salesforce.iosnativetemplate';
var templateOrganization = 'iOSNativeTemplateOrganizationName';
```

Examples of template.js Files

Mobile SDK defines its own templates in the github.com/forcedotcom/SalesforceMobileSDK-Templates repo. Each template directory includes a `template.js` file. Templates include:

- `iOSNativeTemplate` (forceios only)

- `iOSNativeSwiftTemplate` (forceios only)
- `ReactNativeTemplate`
- `HybridLocalTemplate`
- `HybridRemoteTemplate`
- `AndroidNativeTemplate` (forcedroid only)
- `AndroidNativeKotlinTemplate` (forcedroid only)

These templates are "bare bones" projects used by the Mobile SDK npm scripts to create apps; hence, their level of complexity is intentionally low. If you're looking for more advanced templates, see

- `MobileSyncExplorerReactNative`
- `MobileSyncExplorerSwift`
- `AndroidIDPTemplate`
- `iOSIDPTemplate`

You can get a list of these templates with their repo paths from the `listtemplates` command. All Mobile SDK npm scripts—forcedroid, forceios, forcehybrid, and forcereact—support this command.

Note: Always match the script command to the template. Use iOS-specific templates with *`forceios createWithTemplate`* only, and Android-specific templates with *`forcedroid createWithTemplate`* only. This restriction doesn't apply to hybrid and React native templates.

Define a Basic `template.js` File

The following steps describe the quickest way to create a basic `template.js` file.

1. Copy a `template.js` file from the github.com/forcedotcom/SalesforceMobileSDK-Templates repo to the root of your custom template app repo. Be sure to choose the template that matches the type of app your template should build.
2. For native or React native apps only, update the app name, package name, and organization to reflect your template app.
3. If necessary, update the `appType` and `prepare` settings in the `module.exports` object, as described earlier. Although this step isn't required for this basic example, you might need it later if you create your own `template.js` files.

Restrictions and Guidelines

A few restrictions apply to custom templates.

- The template app can be any valid Mobile SDK app that targets any supported platform and architecture.
- A primary requirement is that the template repo and your local Mobile SDK repo must be on the same Mobile SDK version. You can use git version tags to sync both repos to a specific earlier version, but doing so isn't recommended.
- Always match the script command to the template. Use iOS-specific templates with *`forceios createWithTemplate`* only, and Android-specific templates with *`forcedroid createWithTemplate`* only. This restriction doesn't apply to hybrid and React native templates.

Creating an iOS Swift Project Manually

If you prefer not to use forceios or CocoaPods, you can create Mobile SDK apps manually in Xcode.

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

The following tutorial shows two ways to create a native iOS Swift project:

- Borrowing source files from the Mobile SDK Swift template app
- Using Swift Package Manager.

Instructions fall into four sections that must be performed in the sequence shown:

1. Create an Xcode Swift Project on page 46
2. Add Mobile SDK Libraries to Your Project on page 47
3. Configure Your Project's Build Settings on page 47
4. Choose an option:
   - Option 1: Import Mobile SDK Template Files on page 49
   - Option 2: Add Mobile SDK Setup Code Manually on page 51

All sections support Xcode 12.4 and Xcode 13.

IN THIS SECTION:

Create an Xcode Swift Project

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

Summary of Steps

In Xcode, create an **App** project that uses **Swift** language, the **SwiftUI** interface, and the **UIKit App Delegate** life cycle.

Create an Empty Xcode Project—Details

1. In Xcode, select **File** > **New** > **Project**. Or, from the Xcode Welcome screen, select **Create a new Xcode project**.
2. On the iOS tab, select **App**, then click **Next**.
3. Assign the following values:
   - **Product Name:** MyMobileSDKApp
   - **Team:** A team associated with your Apple Developer account
   - **Organization Identifier:** A reverse DNS name, such as `com.acme.apps`
   - **Interface: SwiftUI**

- **Life Cycle: UIKit App Delegate** (This step is not included in Xcode 13 wizards.)
- **Language: Swift**

4. Accept defaults for **Use Core Data** and **Include Tests**, then click **Next**.
5. Choose a directory for your project, then click **Create**.
6. In Xcode, select a recent iPhone model as the active scheme and click **Run**. The app displays only a blank white screen with the words "Hello, world!"
7. Click **Stop**.

Add Mobile SDK Libraries to Your Project

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

Prerequisites

1. Create an Xcode Swift Project on page 46

Add the Mobile SDK Swift Package

For general instructions on adding package dependencies to your app, see Apple's documentation: Adding package dependencies to your app.

To add the Mobile SDK Swift package, use these configurations in Xcode.

- Package repository URL: https://github.com/forcedotcom/SalesforceMobileSDK-iOS-SPM.
- Version: 11.0.1 (or any version thereafter).

Configure Your Project's Build Settings

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

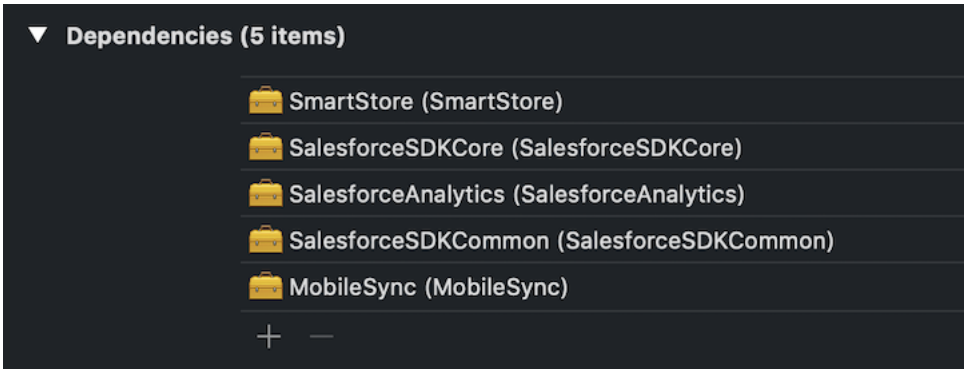
Prerequisites

1. Create an Xcode Swift Project on page 46
2. Add Mobile SDK Libraries to Your Project on page 47

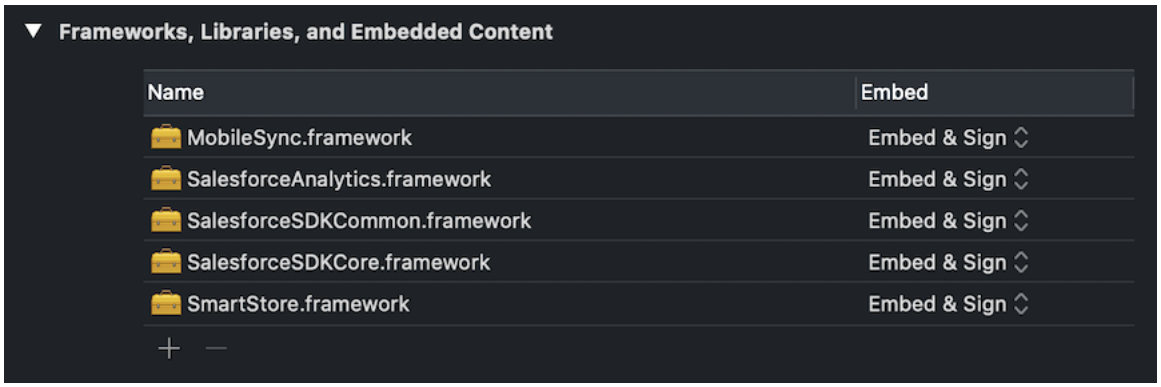
Customize Your Project's Build Configuration

1. In Project Navigator, select the top-level project.
2. In the Project Editor, select **Build Phases**.
3. Under **Dependencies**, click **Add Items** (+).
4. Use Command+Click to multi-select the gold briefcase icons for all five Mobile SDK libraries, then click **Add**.

   When you've finished, you see the following:



5. In the Project Editor, select **General**.
6. Scroll to **Frameworks, Binaries, and Embedded Content**.
7. Click **Add items** (+) and add the following frameworks:
   - `SalesforceSDKCommon.framework`
   - `SalesforceAnalytics.framework`
   - `SalesforceSDKCore.framework`
   - `SmartStore.framework`
   - `MobileSync.framework`



Important: Be sure to add each framework only once. If you accidentally add duplicates, select the extraneous ones and click **Remove items** (-).

8. Run a test build.

Note:

- The result of this build isn't a functional Mobile SDK app. You're only checking to see if the libraries built properly. When the app runs, you still see just a blank white screen that says "Hello, world!"
- If you get a warning about a missing image file:
  a. In Terminal, navigate to `<your selected local path>/MyMobileSDKApp/SalesforceMobileSDK-iOS/`
  b. Run `./install.sh`

9. In Xcode, click **Stop**.

At this point, Mobile SDK is available in your app, but your app can't yet use it. Using Mobile SDK requires coding to initialize the SDK and integrate its services into your app's sessions. To finish setting up Mobile SDK, you have a choice of two options:

Option 1: Import Mobile SDK Template Files on page 49
This option is the quicker, easier route. In your project you remove certain Xcode template files and add files from the iOS native Swift template. Voilà! Your app is ready to use Mobile SDK.

Option 2: Add Mobile SDK Setup Code Manually on page 51
This hands-on coding option gives you a front-row seat to the code changes required for Mobile SDK integration. Along the way, you also might discover some useful customization opportunities. This option doesn't ask you to import template files into your project. However, you'll still benefit from copying a few boilerplate functions found in the iOS native Swift template.

Option 1: Import Mobile SDK Template Files

In this option, you replace files from the Xcode Swift template with files from the Mobile SDK native Swift template—work that otherwise would be done by forceios. A few drags and drops, and the app is ready to connect to Salesforce and display org data.

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

Prerequisites

1. Create an Xcode Swift Project on page 46
2. Add Mobile SDK Libraries to Your Project on page 47
3. Configure Your Project's Build Settings on page 47

Add Mobile SDK Code Assets

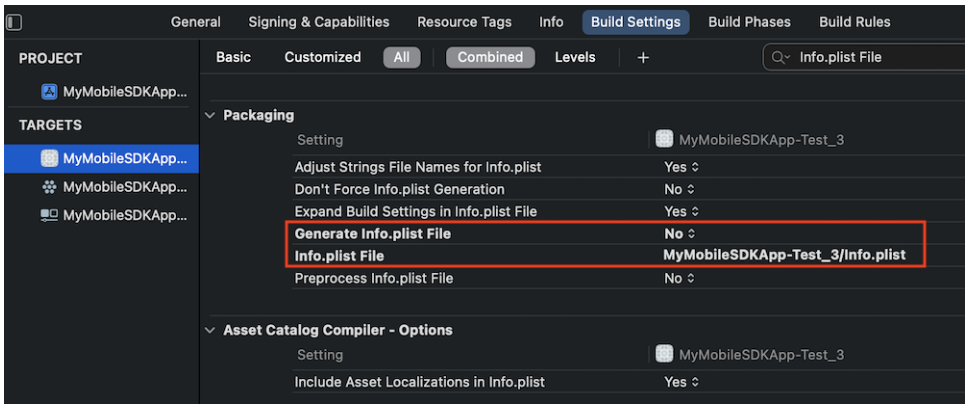
1. In Xcode, open the `MyMobileSDKApp/MyMobileSDKApp` project folder.
2. Select all files *except* the `Assets.xcassets` and `Preview Content` folders.
3. Control-Click and select **Delete**.
4. Select **Move to Trash**.
5. In Finder, navigate to your SalesforceMobileSDK-Templates clone and expand its `/iOSNativeSwiftTemplate/iOSNativeSwiftTemplate/` subfolder.
6. Drag the following Swift template files into Xcode's Project Navigator. Drop them in your project's `MyMobileSDKApp/MyMobileSDKApp/` folder.
   - `AccountsListModel.swift`
   - `AccountsListView.swift`
   - `AppDelegate.swift`
   - `bootconfig.plist`
   - `Bridging-Header.h`
   - `ContactDetailModel.swift`
   - `ContactDetailsView.swift`
   - `ContactsForAccountListView.swift`
   - `ContactsForAccountModel.swift`

- `Info.plist`
- `InitialViewController.swift`
- `main.swift`
- `iOSNativeSwiftTemplate.entitlements`
- `SceneDelegate.swift`
- `userstore.json`
- `usersyncs.json`

7. Select the following:

Option	Value
Destination:	**Copy items if needed**
Added folders:	**Create folder references**
Add to targets:	**MyMobileSDKApp**

8. If Xcode asks for permission to create an Objective-C bridging header, click **Don't Create**.
9. In your Project settings, select **Build Phases**.
10. Under Copy Bundle Resources, select `Info.plist` if it's present and click **Remove Items** (-).
11. *Xcode 13 only*:
    - In **Build Settings**, search for "Info.plist File".
    - Set **Generate Info.plist File** to **No**.
    - Set **Info.plist File** to "MyMobileSDKApp/Info.plist".



For more information on the template files, see Overview of Application Flow.

Build Your New Mobile SDK App

1. Select your top-level project.
2. Your project is ready to build. Select **Product** > **Run**.
3. If the Salesforce login screen appears, you're done!

Your new project is ready for customization. You can now add your own assets, customize the `AppDelegate.swift` and `SceneDelegate.swift` classes, and do as you like with the Accounts and Contacts functionality.

Important: Before posting your app to the App Store, be sure to update the `remoteAccessConsumerKey` and `oauthRedirectURI` in the `bootconfig.plist` file with settings from your connected app. See Get Started with Native iOS Development.

Option 2: Add Mobile SDK Setup Code Manually

If you prefer the freedom of writing all your code from scratch, you can create a project without copying Mobile SDK template files into your workspace. You can—and should—consult the template code to pick up boilerplate implementations for certain features.

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

The procedure here creates a basic Mobile SDK Swift app that implements an `AppDelegate` - `SceneDelegate` flow.

When you insert Mobile SDK into an iOS project without importing template files, you're responsible for reconfiguring the app lifecycle according to Mobile SDK guidelines. For example, here are some required tasks:

- Configure `SalesforceApp` as the main class
- Initialize Mobile SDK
- Call Mobile SDK to prompt for customer login and handle user switching

For a few optional features, Mobile SDK provides specific customizations that you can drop into your code.

As benefits, you have some additional flexibility to suit your development needs:

- Code in Swift (recommended)

  or

  Objective-C (to support older apps; not described in this procedure)

- Use `AppDelegate` for app-level initialization, and `SceneDelegate` to handle all UI lifecycle events. This path is recommended; it supports multiple windows in iPadOS, clarifies multi-scene handling, and separates app-level and scene-level code.

  or

  Use `AppDelegate` alone to handle app initialization and UI lifecycle events. This path is an option only for single-window apps that don't require multiple window support in iPadOS. If you choose this route, it's up to you to find places in `AppDelegate` for all Mobile SDK code that's added here in `SceneDelegate`. See the iOS Native Template for examples.

Note: These instructions provide minimal steps for enabling Mobile SDK in a runnable Swift app. At the end of this procedure, you can find additional sections on setting up optional features such as push notifications and IDP apps.

Prerequisites

1. Create an Xcode Swift Project on page 46
2. Add Mobile SDK Libraries to Your Project on page 47
3. Configure Your Project's Build Settings on page 47

Add or Edit Info.plist

Mobile SDK uses the project's main `Info.plist` file to set the default Salesforce login host. In Xcode 12.4, the app wizard creates the `Info.plist` file for you. In Xcode 13, the app wizard for the SwiftUI interface doesn't create the file, but you can copy one from a Mobile SDK template app.

Xcode 12.4

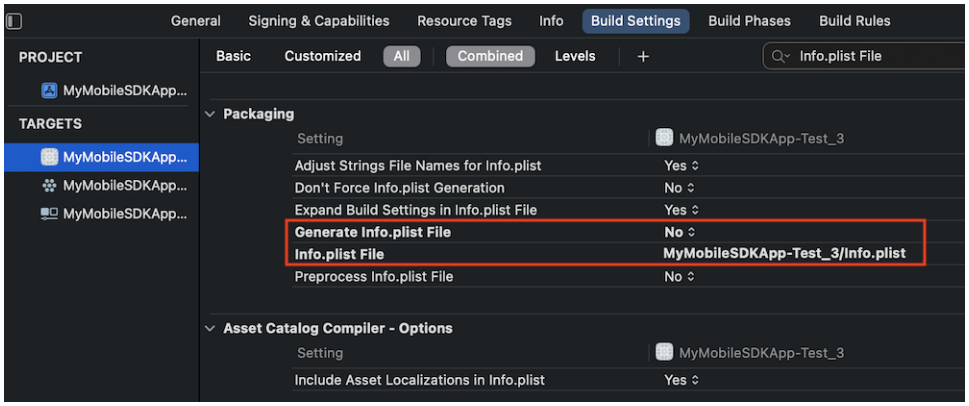
1. In the Project Navigator, control-click **MyMobileSDKApp** > **MyMobileSDKApp** > **Info.plist** and select **Open As** > **Source Code**.
2. Add the following key-value pair to the top-level dictionary:

```
<key>SFDCOAuthLoginHost</key>
<string>login.salesforce.com</string>
```

Xcode 13

Copy the Info.plist file to your project from the iOS Native Swift template. When the file is in place, adjust your Project Settings as follows:

1. In your Project settings, select **Build Phases**.
2. Under Copy Bundle Resources, select `Info.plist`, if it's present, and click **Remove Items** (-).
3. In **Build Settings**, search for "Info.plist File".
4. Set **Generate Info.plist File** to **No**.
5. Set **Info.plist File** to the file's path in your project (for example, "MyMobileSDKApp/Info.plist".)



Add bootconfig.plist

Mobile SDK looks for a `bootconfig.plist` file to help your app start with the correct connected app values. You can copy this file to your project from a Mobile SDK iOS template project, but be sure to reset the values to your app's preferences—especially `remoteAccessConsumerKey` and `oauthRedirectURI`.

1. In an external text editor, create an empty text file.
2. Copy the following text into the new file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
```

```
<dict>
    <key>remoteAccessConsumerKey</key>

<string>3MVG9Iu66FKeHhINkB1l7xt7kR8czFcCTUhgoA8Ol2Ltf1eYHOU4SqQRSEitYFDUpqRWcoQ2.dBv_a1Dyu5xa</string>

    <key>oauthRedirectURI</key>
    <string>testsfdc:///mobilesdk/detect/oauth/done</string>
    <key>oauthScopes</key>
    <array>
        <string>web</string>
        <string>api</string>
    </array>
    <key>shouldAuthenticate</key>
    <true/>
</dict>
</plist>
```

3. Save the file as `bootconfig.plist` in your MyMobileSDKApp project's local disk directory.
4. In Xcode's Project Explorer, control-click **MyMobileSDKApp** > **MyMobileSDKApp** and select **Add Files to "MyMobileSDKApp"**.
5. Select `bootconfig.plist`.
6. To do now (if your connected app is available) or before you release your app:
   a. Open `bootconfig.plist` in the Xcode editor.
   b. Replace the `remoteAccessConsumerKey` value with the Consumer Key from your org's connected app.
   c. Replace the `oauthRedirectURI` value with the Callback URL from your org's connected app.

Add a main.swift File

Mobile SDK requires its own `UIApplication`-based object, `SFApplication`, to monitor user event notifications. This class must be instantiated when iOS first initializes the main application object. To use this class instead of the default `UIApplication` class, you add a custom `main.swift` file to your project. You can then call the `UIApplicationMain` constructor with custom arguments.

1. In your project, control-click **MyMobileSDKApp** > **MyMobileSDKApp** and select **New File**.
2. Select **Swift File**, name it "main.swift", and then click **Create**. Xcode creates a file with one line:

   ```
   import Foundation
   ```

3. Import `SalesforceSDKCore`.

   ```
   import Foundation
   import SalesforceSDKCore
   ```

4. Create an instance of `UIApplicationMain`, passing in
   - "SFApplication" for the `principalClassName` argument
   - "AppDelegate" for the `delegateClassName` argument

   ```
   UIApplicationMain(
       CommandLine.argc,
       CommandLine.unsafeArgv,
   ```

```
    NSStringFromClass(SFApplication.self), //principalClassName
    NSStringFromClass(AppDelegate.self)    //delegateClassName
)
```

See **UIApplicationMain(_:_:_:_:)** in the Apple Developer documentation.

Customize or Create the AppDelegate Class

If you tell the Xcode 12.4 app wizard to create a UIKit App Delegate lifecycle, it creates `AppDelegate` and `SceneDelegate` classes. The Xcode 13 app wizard doesn't offer a lifecycle choice and doesn't create an `AppDelegate` class. However, you can create the basic minimal version from scratch using the following instructions.

1. **Xcode 12.4:** Start with the `AppDelegate` class in your project, keeping the three standard lifecycle functions.

   Xcode 13:

   a. In your project's **MyMobileSDKApp** > **MyMobileSDKApp** folder, control-click the `MyMobileSDKAppApp.swift` file and click **Delete**.

   b. Click **Move to Trash**.

   c. Control-click **MyMobileSDKApp** > **MyMobileSDKApp** and select **New File**.

   d. Select **Swift File**, name it "AppDelegate", and then click **Create**. Xcode creates a file with one line:

   ```
   import Foundation
   ```

2. Paste the following boilerplate implementation of the `AppDelegate` life cycle methods.

```
import Foundation

@main
class AppDelegate : UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    // MARK: UISceneSession Lifecycle
    func application(_ application: UIApplication, configurationForConnecting
        connectingSceneSession: UISceneSession,
        options: UIScene.ConnectionOptions) -> UISceneConfiguration {
        // Called when a new scene session is being created.
        // Use this method to select a configuration to create the new scene with.
        return UISceneConfiguration(name: "Default Configuration",
            sessionRole: connectingSceneSession.role)
    }

    func application(_ application: UIApplication, didDiscardSceneSessions
        sceneSessions: Set<UISceneSession>) {
        // Called when the user discards a scene session.
        // If any sessions were discarded while the application was not running,
        // this will be called shortly after
        // application:didFinishLaunchingWithOptions.
        // Use this method to release any resources that were specific to the
        // discarded scenes, as they will not return.
    }
```

```
    // MARK: - App delegate lifecycle
    func application(_ application: UIApplication, didFinishLaunchingWithOptions
        launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        // If you wish to register for push notifications, uncomment the line below.
        // Note that, if you want to receive push notifications from Salesforce,
        // you will also need to implement the
        // application(application, didRegisterForRemoteNotificationsWithDeviceToken)
        // method (below).

        // self.registerForRemotePushNotifications()
        return true
    }
}
```

3. Remove the `@main` attribute above the `AppDelegate` class declaration. The `main.swift` file has assumed the role of main.
4. Import all Mobile SDK modules.

```
import Foundation
import MobileSync
```

5. At the top of the class, override the `init()` method to initialize Mobile SDK. For example, to initialize all Mobile SDK libraries, call `initializeSDK()` on `MobileSyncSDKManager`:

```
override init() {
    super.init()
    MobileSyncSDKManager.initializeSDK()
}
```

Add an InitialViewController Class

To access Salesforce data, your app's users must authenticate with a Salesforce org. Salesforce sends a login screen to your app to collect the user's credentials and then returns OAuth tokens to your app.

Your app must include a view controller whose sole purpose is to host the Salesforce login screen. When no customer is logged in, your app sets this view controller as the root view. Traditionally, Mobile SDK apps name this view controller class `InitialViewController`. It requires only a stub implementation.

1. In your project, control-click **MyMobileSDKApp** > **MyMobileSDKApp** and select **New File**.
2. Select **Swift File**, name it "InitialViewController", and then click **Create**.
3. Add the following body:

```
import Foundation
import UIKit

class InitialViewController: UIViewController {
}
```

Customize or Create the SceneDelegate Class

Use the `SceneDelegate` class to handle Salesforce logins, user switching, and root view instantiation. Login is required when no user is logged in—when the app originally launches, and when the current user logs out. User changes occur between users who have logged in previously and remain authenticated. Mobile SDK `AuthHelper` functions simplify how these cases are handled:

- `AuthHelper.loginIfRequired(_:)` decides for you whether login is needed
- `AuthHelper.registerBlock(forCurrentUserChangeNotifications:)` listens for user change events.

These functions let you determine what happens after the login or change occurs. Typically, you respond by setting your app's main view as the root view.

In the scene lifecycle, specific junctures are ideal for each of these `AuthHelper` calls. For example, register the block that handles user changes only one time, when the scene is first initialized. Also, call `AuthHelper.loginIfRequired(_:_:)` whenever the scene comes to the foreground. If the current user remains logged in, Mobile SDK merely executes your completion block.

At app startup, you set `InitialViewController` as the first root view controller. This view hosts the Salesforce login screen and, after a successful login, is replaced by your app's root view controller. At runtime, if the current user logs out, you dismiss the current root view controller and replace it with `InitialViewController`. In both cases, after a new user logs in, you dismiss `InitialViewController` and set your app's first view as the root view controller.

Xcode 13 only: In Xcode 13, the SwiftUI interface no longer uses `SceneDelegate`. However, iOS still supports it, and Mobile SDK currently uses it. Follow these steps to create it.

1. In your project, control-click **MyMobileSDKApp** > **MyMobileSDKApp** and select **New File**.
2. Select **Swift File**, name it "SceneDelegate", and then click **Create**.
3. Copy the following implementation into your new file.

```
import UIKit
import MobileSync
import SwiftUI

class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?


    func scene(_ scene: UIScene, willConnectTo session: UISceneSession,
        options connectionOptions: UIScene.ConnectionOptions) {
        // Use this method to optionally configure and attach the
        // UIWindow `window` to the provided UIWindowScene `scene`.
        // If using a storyboard, the `window` property will automatically
        // be initialized and attached to the scene. This delegate does not
        // imply the connecting scene or session are new
        // (see `application:configurationForConnectingSceneSession` instead).

        // Create the SwiftUI view that provides the window contents.
        let contentView = ContentView()

        // Use a UIHostingController as window root view controller.
        if let windowScene = scene as? UIWindowScene {
            let window = UIWindow(windowScene: windowScene)
            window.rootViewController = UIHostingController(rootView: contentView)
            self.window = window
            window.makeKeyAndVisible()
```

```
        }
    }

    func sceneDidDisconnect(_ scene: UIScene) {
        // Called as the scene is being released by the system.
        // This occurs shortly after the scene enters the background,
        // or when its session is discarded.
        // Release any resources associated with this scene that can be re-created
        // the next time the scene connects.
        // The scene may re-connect later, as its session was not necessarily
        // discarded (see `application:didDiscardSceneSessions` instead).
    }

    func sceneDidBecomeActive(_ scene: UIScene) {
        // Called when the scene has moved from an inactive state to an active state.
        // Use this method to restart any tasks that were paused (or not yet started)
        // when the scene was inactive.
    }

    func sceneWillResignActive(_ scene: UIScene) {
        // Called when the scene will move from an active state to an inactive state.
        // This may occur due to temporary interruptions (ex. an incoming phone call).
    }

    func sceneWillEnterForeground(_ scene: UIScene) {
        // Called as the scene transitions from the background to the foreground.
        // Use this method to undo the changes made on entering the background.
    }

    func sceneDidEnterBackground(_ scene: UIScene) {
        // Called as the scene transitions from the foreground to the background.
        // Use this method to save data, release shared resources, and store enough
        // scene-specific state information
        // to restore the scene back to its current state.
    }
}
```

Xcode 12.4 and Xcode 13:

1. At the top of your project's `SceneDelegate` class, be sure that you've declared a `UIWindow` variable:

```
class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?
```

2. At the bottom of the SceneDelegate class body, implement a private function named `resetViewState(_:)` that takes an escaping closure as its sole parameter. This function dismisses the current root view controller and then executes the given closure:

```
func resetViewState(_ postResetBlock: @escaping () -> ()) {
    if let rootViewController = self.window?.rootViewController {
        if let _ = rootViewController.presentedViewController {
            rootViewController.dismiss(animated: false,
                completion: postResetBlock)
            return
        }
```

```
            self.window?.rootViewController = nil
        }
        postResetBlock()
    }
```

3. Implement a private function named `setupRootViewController()` that launches your app's custom views. For this simple exercise, you can set the root view to the SwiftUI `ContentView` provided by the Xcode template.

```
func setupRootViewController() {
    self.window?.rootViewController = UIHostingController(rootView:
        ContentView())
}
```

Note: The Xcode template's `ContentView` requires no additional setup or cleanup. For a more elaborate example, see `setupRootViewController(_:)` in the MobileSyncExplorerSwift template.

4. In the `scene(_:willConnectTo:options:)` function, provide code that sets up the window scene. Replace all existing code with the following lines:

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession,
    options connectionOptions: UIScene.ConnectionOptions) {

    guard let windowScene = (scene as? UIWindowScene) else { return }
    self.window = UIWindow(frame: windowScene.coordinateSpace.bounds)
    self.window?.windowScene = windowScene


}
```

5. Register a callback block for user change events. Using the private functions you added, provide a closure that resets the view state and then presents your app's root view.

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession,
    options connectionOptions: UIScene.ConnectionOptions) {

    guard let windowScene = (scene as? UIWindowScene) else { return }
    self.window = UIWindow(frame: windowScene.coordinateSpace.bounds)
    self.window?.windowScene = windowScene

    // Define a response to user change events
    AuthHelper.registerBlock(forCurrentUserChangeNotifications: {
        self.resetViewState {
            self.setupRootViewController()
        }
    })
}
```

6. Implement a private function named `initializeAppViewState()` that sets `InitialViewController` as the root view. Notice that this function first makes sure that the app is on the main UI thread. If the app is on some other thread, it switches to the main thread and calls itself recursively.

```
func initializeAppViewState() {
    if (!Thread.isMainThread) {
        DispatchQueue.main.async {
```

```
            self.initializeAppViewState()
        }
        return
    }

    self.window?.rootViewController =
        InitialViewController(nibName: nil, bundle: nil)
    self.window?.makeKeyAndVisible()
}
```

7. In the `sceneWillEnterForeground(_:)` listener function:
   a. To set the root view to `InitialViewController`, call `initializeAppViewState()`.
   b. Call `AuthHelper.loginIfRequired(_:)` with a closure that sets the root view controller to your app's main view.

```
func sceneWillEnterForeground(_ scene: UIScene) {
    // Called as the scene transitions from the background to the foreground.
    // Use this method to undo the changes made on entering the background.
    self.initializeAppViewState()
    AuthHelper.loginIfRequired {
        self.setupRootViewController()
    }
}
```

Build Your New Mobile SDK App

1. In Xcode, select **Product** > **Run** (R).
2. If the Salesforce login screen appears, you're done!

Your new project is ready for customization. You can now add your own assets and set up your scene flow. You can also add optional push notification and IDP features, as follows.

Important: Another reminder! Before posting your app to the App Store, be sure to update the `remoteAccessConsumerKey` and `oauthRedirectURI` in the `bootconfig.plist` file with settings from your connected app. See Get Started with Native iOS Development.

Add Push Notification Support

1. To support Salesforce push notifications, add code to register and handle them. You can copy boilerplate implementations of the following methods from the `AppDelegate` class of the Mobile SDK iOSNativeSwiftTemplate app:
   - `registerForRemotePushNotifications()`
   - `application(_:didRegisterForRemoteNotificationsWithDeviceToken:)`—**Be sure to uncomment the one line of implementation code!**
   - `didRegisterForRemoteNotifications(_:)`
   - `application(_:didFailToRegisterForRemoteNotificationsWithError:)`
2. Salesforce uses encryption on notifications, which requires you to implement a decryption extension and then to request authorization through the `UNUserNotificationCenter` object. See Code Modifications (iOS) on page 399.

Add Identity Provider Support

Mobile SDK supports configuration of identity provider (IDP) apps. See Identity Provider Apps on page 442. If you've implemented IDP functionality, you enable it by customizing `AppDelegate` and `SceneDelegate`.

1. In your `AppDelegate` class, add the following boilerplate functions from the `AppDelegate` class of the Mobile SDK iOSNativeSwiftTemplate app:
   - `application(_:open:_:)`
   - `enableIDPLoginFlowForURL(_:_:)`
2. In your `SceneDelegate` class, add the following functions.

```
func scene(_ scene: UIScene, openURLContexts URLContexts: Set<UIOpenURLContext>) {
    if let urlContext = URLContexts.first {
        self.enableIDPLoginFlowForURLContext(urlContext, scene: scene)
    }
}

func enableIDPLoginFlowForURLContext(_ urlContext: UIOpenURLContext,
    scene: UIScene) -> Bool {
    return UserAccountManager.shared.handleIdentityProviderResponse(
        from: urlContext.url, with: [UserAccountManager.IDPSceneKey:
            scene.session.persistentIdentifier])
}
```

Customize the Pre-Login Screen

You can use the `InitialViewController` class to cosmetically customize the login view before the login screen appears. Here's an example that brands the pre-login screen with your app's name, using the default Salesforce background color. If you use this class to create resources that can be recreated, be sure to implement the `didReceiveMemoryWarning()` override to dispose of them. See the Xcode UIViewController documentation.

```
import Foundation
import UIKit
import SalesforceSDKCore.UIColor_SFColors

class InitialViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        self.view.backgroundColor = UIColor.salesforceSystemBackground
        let label = UILabel()
        label.translatesAutoresizingMaskIntoConstraints = false
        guard let info = Bundle.main.infoDictionary,
            let name = info[kCFBundleNameKey as String] else { return }
        label.font = UIFont.systemFont(ofSize: 29)
        label.textColor = UIColor.black
        label.text = name as? String
        self.view.addSubview(label)
        label.centerXAnchor.constraint(equalTo: self.view.centerXAnchor).
            isActive = true
        label.centerYAnchor.constraint(equalTo: self.view.centerYAnchor).
            isActive = true
```

```
        // Do any additional setup after loading the view,
        // typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

Include SmartStore and Mobile Sync Configuration Files

Offline configuration files let you define your SmartStore and Mobile Sync setups in static XML files instead of code. These files must be named userstore.json and usersyncs.json and be referenced in your Xcode project. You can then load them by calling the following methods:

```
// Setup store based on config userstore.json
MobileSyncSDKManager.shared.setupUserStoreFromDefaultConfig()
// Setup syncs based on config usersyncs.json
MobileSyncSDKManager.shared.setupUserSyncsFromDefaultConfig()
```

Your main consideration for these files is to be sure to load them only one time in each session. In a `SceneDelegate` flow, for example, loads the config files when

- Salesforce login is required, or
- When a change of users occurs

The Mobile SDK native Swift template keeps things simple by calling the two setup functions whenever it sets the root view controller. This task falls to a private function, `setupRootViewController()`, that in turn is called as follows:

- In `sceneWillEnterForeground(_:)`, where it's called in a block that runs after a Salesforce login occurs.
- In `scene(_:willConnectTo:options:)`, where it's called in a block that handles current user change notifications.

Mobile SDK defines a utility class, `AuthHelper`, that coordinates the internal moving parts for login and user change events. For a complete example, see the SceneDelegate class in the Mobile Sync Explorer Swift template.

Note: If your app defers login, where and when you call `AuthHelper.loginIfRequired(_:)` is up to you. In all cases, be sure to precede the call by setting the root view controller to `IntialViewController`, as shown here.

Use CocoaPods with Mobile SDK

CocoaPods provides a convenient mechanism for merging Mobile SDK modules into existing Xcode projects. The steps in this article guide you through manually setting up CocoaPods in a Mobile SDK iOS app. If you created your app with forceios 4.0 or later, you get the CocoaPods setup automatically. In that case, you don't have to perform the steps in this article—you only have to install CocoaPods software, and forceios does the rest. If you're creating apps some other way, use this article if you plan to import Mobile SDK modules through CocoaPods.

In Mobile SDK 4.0 and later, forceios uses CocoaPods to create projects. Developers can also use CocoaPods manually to add Mobile SDK to existing iOS apps.

If you're unfamiliar with CocoaPods, start by reading the documentation at www.cocoapods.org.

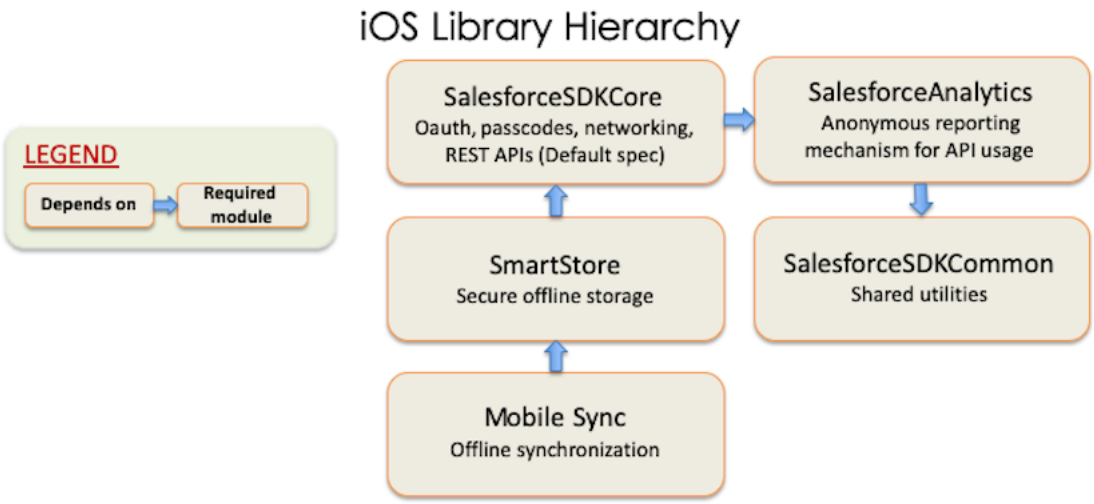
Note: The forceios npm utility is provided as an optional convenience. CocoaPods, node.js, and npm are required for forceios but are not required for Mobile SDK iOS development. To learn how to create Mobile SDK iOS native projects without forceios, see Creating an iOS Swift Project Manually.

Mobile SDK provides CocoaPods pod specifications, or *podspecs*, for each Mobile SDK module.

- `SalesforceSDKCore`—Implements OAuth, passcodes, networking, and REST APIs.
- `SmartStore`—Implements secure offline storage. Depends on `SalesforceSDKCore`.
- `Mobile Sync`—Implements offline synchronization. Depends on `SmartStore`.
- `SalesforceAnalytics`—Implements a reporting mechanism that sends Salesforce anonymous statistics on Mobile SDK feature usage and popularity.
- `SalesforceSDKCommon`—Utilities shared throughout the SDK.

Mobile SDK also consumes FMDB, an external module, through CocoaPods.

The following chart shows the dependencies between specs. In this chart, the arrows point from the dependent specs to their dependencies.



If you declare a pod, you automatically get everything in that pod's dependency chain. For example, by declaring a pod for Mobile Sync, you automatically get the `SmartStore` and `SalesforceSDKCore` pods. This shortcut applies only to production pods.

You can access all versions of the Mobile SDK podspecs in the github.com/forcedotcom/SalesforceMobileSDK-iOS-Specs repo. You can also get the current version from the github.com/forcedotcom/SalesforceMobileSDK-iOS repo.

To use CocoaPods with the current Mobile SDK release, follow these steps.

1. Be sure you've installed the `cocoapods` Ruby gem as described at www.cocoapods.org. Mobile SDK 11.1 accepts pod versions 1.8 to no declared maximum.
2. In your project's Podfile, add the SalesforceMobileSDK-iOS-Specs repo as a source. Make sure that you put this entry first, before the CocoaPods source path.

```
target 'YourAppName' do
source 'https://github.com/forcedotcom/SalesforceMobileSDK-iOS-Specs.git' # needs to be
 first
source 'https://github.com/CocoaPods/Specs.git'
...
```

3. Reference the Mobile SDK podspec that you intend to merge into your app. For example, to add OAuth and passcode modules to your app, declare the `SalesforceSDKCore` pod in your Podfile. For example:

```
target 'YourAppName' do
source 'https://github.com/forcedotcom/SalesforceMobileSDK-iOS-Specs.git' # needs to be
 first
source 'https://github.com/CocoaPods/Specs.git'

pod 'SalesforceSDKCore'

end
```

   This pod configuration is the minimum for a Mobile SDK app.

4. To add other modules, replace `SalesforceSDKCore` with a different pod declaration. For example, to use Mobile Sync:

```
target 'YourAppName' do
source 'https://github.com/forcedotcom/SalesforceMobileSDK-iOS-Specs.git' # needs to be
 first
source 'https://github.com/CocoaPods/Specs.git'

pod 'MobileSync'

end
```

   Since the `MobileSync` pod depends on `SmartStore` and `SalesforceSDKCore`, you don't need to declare those pods explicitly.

5. (Alternate method) To work with the upcoming release of Mobile SDK, you clone the SalesforceMobileSDK-iOS repo, check out the `dev` branch, and then pull resources from it. In this case, you must declare each pre-release dependency explicitly so you can indicate its repo path. If you omit a dependency declaration, CocoaPods loads its production version.
   a. Clone github.com/forcedotcom/SalesforceMobileSDK-iOS locally at the desired commit.
   b. At the Terminal window, run *`git checkout dev`* to switch to the development branch.
   c. Run *`./install.sh`* in the root directory of your clone.
   d. To each pod call in your Podfile, add a `:path` parameter that points to your clone.

   Here's the previous example repurposed to pull resources from a local clone:

```
target 'YourAppName' do
source 'https://github.com/forcedotcom/SalesforceMobileSDK-iOS-Specs.git' # need to be
 first
source 'https://github.com/CocoaPods/Specs.git'
```

```
# Specify each pre-release pod
pod 'SalesforceSDKCore', :path => '/<path-to-clone-of>/SalesforceMobileSDK-iOS/'
pod 'SalesforceAnalytics', :path => '/<path-to-clone-of>/SalesforceMobileSDK-iOS/'
pod 'SmartStore', :path => '/<path-to-clone-of>/SalesforceMobileSDK-iOS/'
pod 'MobileSync', :path => '/<path-to-clone-of>/SalesforceMobileSDK-iOS/'

end
```

6. In a Terminal window, run *pod install* from your project directory. CocoaPods downloads the dependencies for your requested pods, merges them into your project, and creates a workspace containing the newly merged project.

   Important: After running CocoaPods, always access your project only from the workspace that *pod install* creates. For example, instead of opening `MyProject.xcodeproj`, open `MyProject.xcworkspace`.

7. To use Mobile SDK APIs in your merged app, remember these important tips.

   a. In Objective-C apps, import header files using angle brackets ("<" and ">") rather than double quotes. For example:

   ```
   #import <SalesforceSDKCore/SFRestAPI.h>
   ```

   b. In Swift apps, be sure to specify `use_frameworks!` in your Podfile. Also, in your Swift source files, remember to import modules instead of header files. For example:

   ```
   import SalesforceSDKCore
   ```

Refreshing Mobile SDK Pods

CocoaPods caches its pods in repos stored locally on your machine. If the pod repo gets out of sync with forceios, you can manually update it.

When forceios creates a native app, it prints a list of installed pods and their versions. For example:

```
Installing SalesforceSDKCore (8.0.0)
Installing SalesforceAnalytics (8.0.0)
Installing SmartStore (8.0.0)
Installing MobileSync (8.0.0)
```

You can compare these versions to your forceios version by typing:

```
forceios version
```

If the reported pod versions are older than your forceios version, run the following commands in the Terminal window:

```
pod repo remove forcedotcom
pod setup
```

After setup completes, recreate your app with *forceios create*.

Using Carthage with Mobile SDK (Not Supported)

In Mobile SDK 9.0 and later, Carthage is no longer officially supported.

Developing a Native iOS App

Salesforce Mobile SDK for native iOS provides the tools you need to build apps for Apple mobile devices. Features of the SDK include:

- Swift and Objective-C APIs
- A network library with wrapper methods that make it easy to call Salesforce REST APIs
- Fully implemented OAuth login and passcode protocols
- SmartStore and Mobile Sync libraries for securely managing user data offline

Mobile SDK for native iOS requires you to be proficient in the iOS SDK. You can choose to write your apps in Swift or Objective-C, though we highly recommend using Swift. You also need to be familiar with iOS application development principles and frameworks. If you're a newbie, developer.apple.com/develop/ is a good place to begin learning. See iOS Basic Requirements for additional prerequisites.

In some Mobile SDK interfaces, you're required to override some methods and properties. SDK header files include comments that indicate mandatory and optional overrides.

About Login and Passcodes

To access Salesforce objects from a Mobile SDK app, the customer logs in to a Salesforce org. When the login flow begins, Mobile SDK sends the app's connected app configuration to Salesforce. Salesforce responds by posting a login screen to the mobile device.

Optionally, a Salesforce administrator can set the connected app to require a passcode after login. This setting, for example, requires a backgrounded app to prompt for a passcode when it returns to the foreground. When the connected app requires a mobile app passcode, Mobile SDK 9.2 and later use the device system passcode.

To verify a required passcode, Mobile SDK presents a lock screen that uses the customer's configured verification mode—for example, touch ID, face ID, or direct passcode entry. If no device passcode has been set, Mobile SDK prompts the customer to visit the iOS Settings interface and create one. If the connected app doesn't require a mobile passcode, Mobile SDK skips the passcode verification step. Mobile SDK handles all login and passcode lock screens and the authentication handshake.

Mobile Passcode Policies

Each Mobile SDK app hard-codes a connected app's consumer key and OAuth callback URL from a specific Salesforce org. Mobile SDK honors the configurable passcode requirement in that org's designated connected app. Beginning in version 9.2, Mobile SDK ignores org settings such as PIN length, and instead relies on device configuration. Similarly, incorrect passcode entries are handled according to the standard procedure of the mobile operating system.

Note: Beginning in version 9.2, Mobile SDK ignored the **Lock App After** setting in the org's Connected App, in favor of the device's configuration for locking the device after it's been idle. In version 10.1.1 and later, Mobile SDK again respects the **Lock App After** Connected App setting. When set, the mobile app locks after it has been in the background for longer than the timeout period. Locking occurs when the mobile app is activated. Unlocking the app remains the same.

If a customer uses the app to log into a different org, Mobile SDK can't retrieve the designated connected app settings. Therefore, that customer never encounters the passcode prompt.

Multi-User Behavior of the Lock Screen

When multiple users are logged into the same app on the same device, the lock screen behaves as follows.

1. When resuming an app that requires passcode, the customer is first prompted by a lock screen to authenticate through the mobile operating system.
2. If the customer cancels authentication, **Logout** and **Retry Unlock** buttons appear on the lock screen.

3. The **Logout** button works only for customers that require the lock screen.
4. If the last user that requires the lock screen logs out, Mobile SDK no longer shows the lock screen.

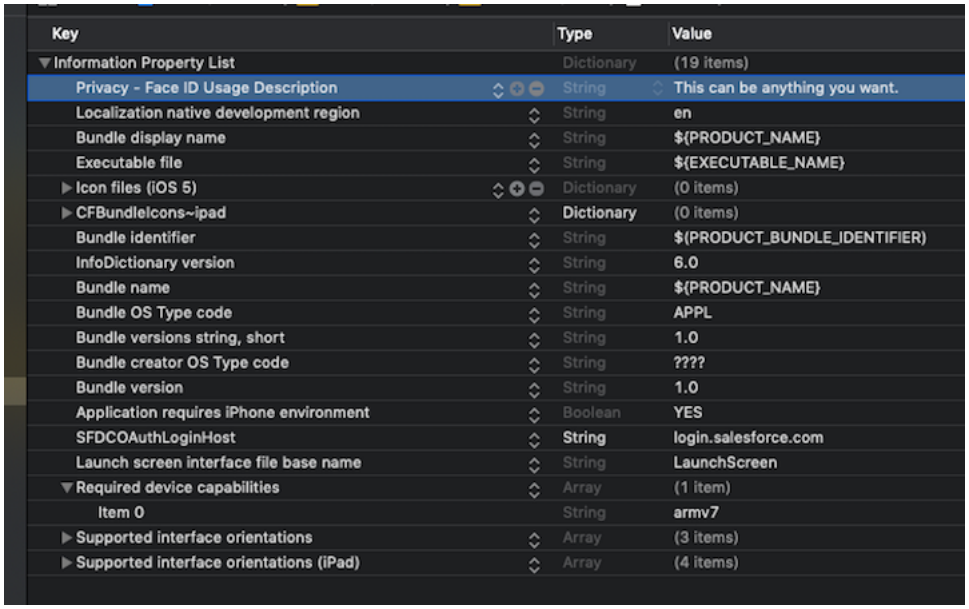
Using Biometric Identification

By default, Mobile SDK supports the use of Touch ID and Face ID for passcode verification. Customers can set their preference for biometric or any other iOS input mode in Settings.

Biometric identification requires the following app configuration.

App Configuration

1. Open your app's information property list (plist) file.
2. Add the following top-level property:
   - Key: **NSFaceIDUsageDescription** (labeled "Privacy - Face ID Usage Description")
   - Value: **<*Enter any string*>** (for example, "Use Face ID to avoid retyping your PIN")



Disabling Biometric Identification

Apps built with Mobile SDK 9.2 and later ignore biometric settings from Salesforce connected apps. Instead, customers can configure the authentication mode themselves in device settings.

See Also

- About PIN Security
- OAuth 2.0 Authentication Flow
- Biometric Authentication on page 420

Using Swift with Salesforce Mobile SDK

Creating Swift Apps

To create Mobile SDK Swift projects, you can assemble the pieces manually, or you can use the `forceios create` command. With forceios, specify `native_swift` as the application type, or simply press `Return`. For example:

```
forceios create
Enter your application type (native, native_swift, leave empty for native_swift): <press RETURN>
Enter your application name: TestSwift
Enter your package name: com.bestapps.swift
Enter your organization name (Acme, Inc.): BestApps, Inc.
Enter output directory for your app (leave empty for the current directory): TestSwift
```

For manual app creation steps, see Creating an iOS Swift Project Manually.

Swift API Naming

In Mobile SDK, Swift methods and classes closely follow the related Objective-C API signatures. However, to facilitate calling Objective-C APIs, Mobile SDK defines Swift-friendly names for most methods and parameters. When Mobile SDK defines a custom Swift name, it appears in the Objective-C header file with the tag `NS_SWIFT_NAME` immediately following the Objective-C method declaration. Here's an example method declaration:

```
- (nullable SFSyncState*) reSyncByName:(NSString*)syncName
                           updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
                           NS_SWIFT_NAME(reSync(named:onUpdate:));
```

With class and protocol names, the declaration comes before the class declaration:

```
NS_SWIFT_NAME(RestClient)
@interface SFRestAPI : NSObject
```

If an Objective-C API is not available in Swift, you'll see the tag `NS_SWIFT_UNAVAILABLE("")` next to the API declaration.

Most Swift class names are simply the Objective-C name without the "SF" prefix, but there are some refinements as well. For example, `SFRestAPI` equates to `RestClient` in Swift. Here are some noteworthy examples.

Objective-C	Swift
`SalesforceSDKManager`	`SalesforceManager`
`SFUserAccountManager`	`UserAccountManager`
`SFRestAPI`	`RestClient`
`SFSmartStore`	`SmartStore`
`SFMobileSync*`	`MobileSync*`

iOS Native Template Apps

Mobile SDK 9.0 updates its default Swift template to iOS 14 standards and maintains its Objective-C template from the previous release.

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

Here's a comparison summary of the forceios default templates.

Swift Template

- iOSNativeSwiftTemplate
- Demonstrates two levels of master-detail navigation.
- Uses SwiftUI for all views
- Demonstrates model-view architecture
- Distributes management of app life-cycle events between `AppDelegate` and `SceneDelegate`
- Handles asynchronous REST responses with various Combine Publisher objects
- Demonstrates offline features using SmartStore and Mobile Sync

Objective-C Template

- iOSNativeTemplate
- Displays a single list of contacts
- Manages app life-cycle events through `AppDelegate`
- Handles asynchronous REST responses in `SFRestDelegate`

You can find both templates in the github.com/forcedotcom/SalesforceMobileSDK-Templates repo. How you use them with forceios is slightly different.

- To select the native Swift application type, press *`Return`* or type *`native_swift`*:

```
$ forceios create
Enter your application type (native_swift or native, leave empty for native_swift):
<Press Return>
...
```

- To select the Objective-C application type, type *`native`*:

```
$ forceios create
Enter your application type (native_swift or native, leave empty for native_swift):
native
...
```

Using Other Templates

Mobile SDK also offers the following specialized iOS templates.

MobileSyncExplorerSwift
A more extended example of Mobile Sync technology. Includes a Recent Contacts widget implementation. Also demonstrates iPad features, including multiple windows and landscape orientation support.

iOSIDPTemplate
For creating identity provider apps with Mobile SDK.

iOSNativeSwiftEncryptedNotificationTemplate
Demonstrates how to process encrypted notifications.

You use these templates with the `forceios createwithtemplate` command. For example, to create an app with the Swift encrypted notifications template:

- ```
  $ forceios createwithtemplate
  Enter URI of repo containing template application:
  iOSNativeSwiftEncryptedNotificationTemplate
  ...
  ```

  Note: (Mobile SDK 8.0 and later) If you're using a template from the SalesforceMobileSDK-Templates repo, you can specify just the template name without the URI path.

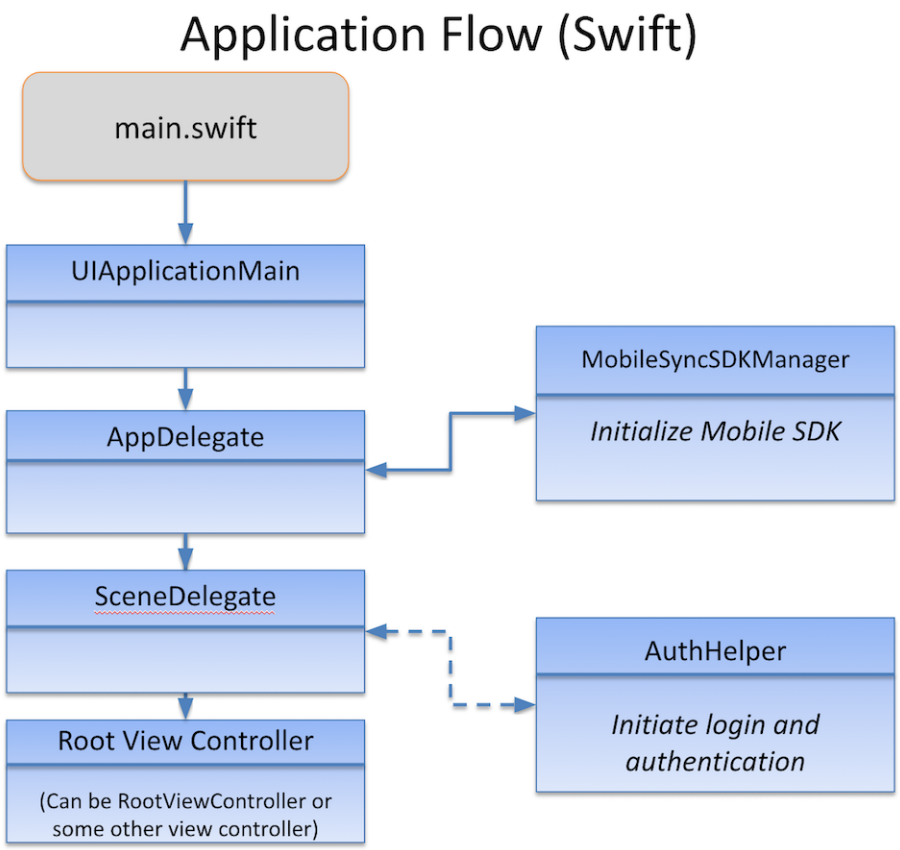
Overview of Application Flow

Native iOS apps for Mobile SDK, whether Objective-C and Swift, follow similar designs. The `main.swift` source file (`main.m` in Objective-C) creates a `UIApplicationMain` object that is the root object for the rest of the application. The `UIApplicationMain` constructor instantiates the Mobile SDK `SFApplication` object and spawns an `AppDelegate` object that manages the application lifecycle.

`AppDelegate` uses a Mobile SDK service object, `SalesforceSDKManager`, to initialize SDK objects. After successful initialization, the Objective-C and Swift flows diverge slightly.

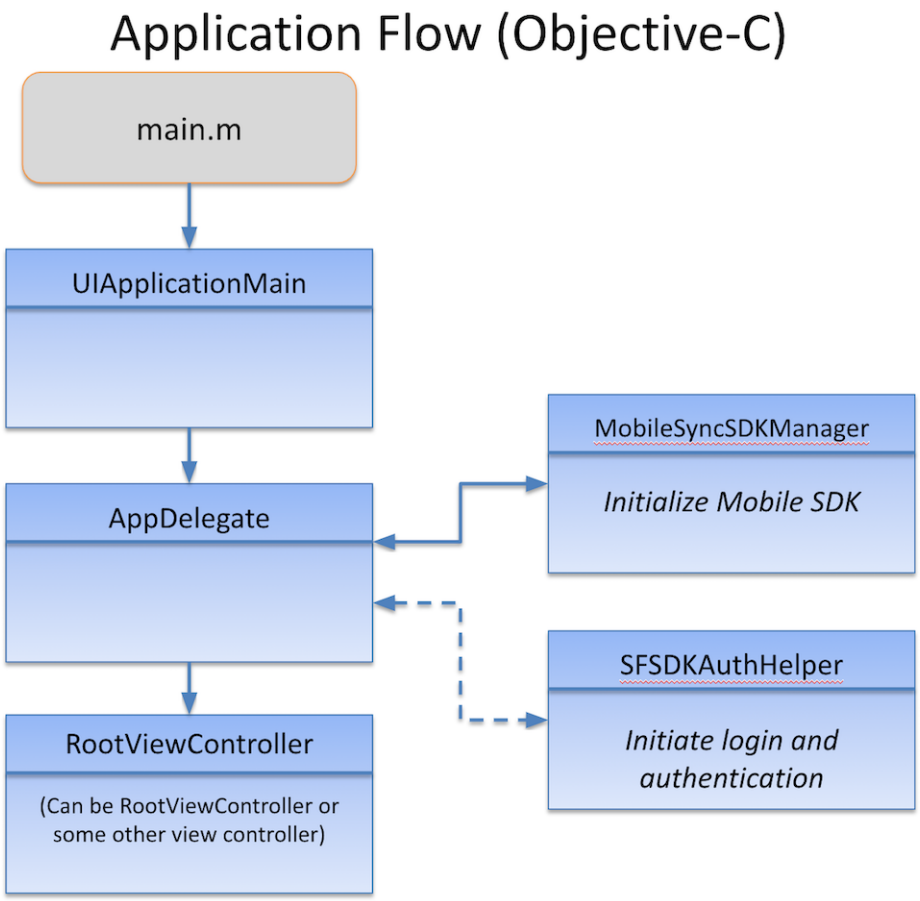
Swift Flow

The native iOS Swift template for Mobile SDK defines four main classes: `AppDelegate`, `InitialViewController`, `SceneDelegate`, and a root view controller. The `AppDelegate` object calls `SalesforceSDKManager` or one of its extensions to initialize the SDK. It then passes control to a `SceneDelegate` instance. When a new scene is about to enter the foreground, `SceneDelegate` loads `InitialViewController`—an empty `UIViewController` object that's a container for the Salesforce login and authorization screens. `SceneDelegate` then calls `AuthHelper` to begin the authentication process if necessary. After authentication succeeds, `SceneDelegate` launches your app by passing control to its custom root view controller. For this template, the root view controller is the `AccountsListView` class.
  ```



Objective-C Flow

The Objective-C native iOS template defines three classes: `AppDelegate`, `InitialViewController`, and `RootViewController`. The `AppDelegate` object calls `SFSDKAuthHelper` to load `InitialViewController` for its first view—the login screen. After authentication succeeds, the `AppDelegate` object displays the view associated with `RootViewController` as the entry point to your app.



Note: The workflow demonstrated by the template app is just an example. You can tailor your `AppDelegate` and supporting classes to enable extra features or refine the workflow. For example, you can postpone Salesforce authentication until it's required. You can retrieve data through REST API calls and display it, launch other views, perform services, and so on. Mobile SDK templates also provide commented-out boilerplate code that you can reinstate to handle push notifications and initialize IDP app services.

SEE ALSO:

SDK Manager Classes

SDK Manager Classes

Swift	Objective-C
`SalesforceManager`	`SalesforceSDKManager`
`SmartStoreSDKManager`	`SmartStoreSDKManager`
`MobileSyncSDKManager`	`MobileSyncSDKManager`
`SalesforceHybridSDKManager`	`SalesforceHybridSDKManager`
`SalesforceReactSDKManager`	`SalesforceReactSDKManager`

The SalesforceManager class performs bootstrap configuration and initializes Mobile SDK components. It also coordinates all processes involved in app launching, including PIN code, OAuth configuration, and other bootstrap processes. In effect, SalesforceManager shields you from having to coordinate the app launch dance yourself.

In addition to SalesforceManager, Mobile SDK provides specialized SDK manager classes. The following SDK manager classes support special flavors of Mobile SDK. Their names are the same for Objective-C and Swift.

- `SmartStoreSDKManager`—subclass of SalesforceManager. For native apps that use SmartStore but not Mobile Sync. Replaces the deprecated `SalesforceSDKManagerWithSmartStore` class.
- `MobileSyncSDKManager`—subclass of `SmartStoreSDKManager`. Provides access to the full range of available Mobile SDK native features.
- `SalesforceHybridSDKManager`—subclass of `MobileSyncSDKManager`. For hybrid Cordova apps only. Includes all available Mobile SDK hybrid features.
- `SalesforceReactSDKManager`—subclass of `MobileSyncSDKManager`. For React Native apps only. Includes all available Mobile SDK React Native features.

All Mobile SDK apps use SalesforceManager or one of its subclasses to initialize Mobile SDK. To use one of these classes, you call its `initializeSDK` method in the initialization of your `AppDelegate` class. Apps normally don't call SDK manager methods outside of `AppDelegate`. Apps created with the Mobile SDK native templates use an instance of `MobileSyncSDKManager`. The effect of this choice is that Mobile SDK internally configures `SalesforceSDKManager` to use `MobileSyncSDKManager` as its instance class.

Swift Example

In the `AppDelegate` class, the Swift template calls `initializeSDK()` on the `MobileSyncSDKManager` class:

```
import Foundation
import UIKit
import MobileSync


class AppDelegate : UIResponder, UIApplicationDelegate
{
    var window: UIWindow?

    override
    init()
    {
        super.init()
        MobileSyncSDKManager.initializeSDK()
...
```

The `initializeSDK()` call ensures that the correct pieces are in place for using SmartStore, Mobile Sync, and everything in the core SDK. To use a different SDK manager object, import its module and replace `MobileSyncSDKManager` in your code with the name of the appropriate manager class.

Objective-C Example

You use the `MobileSyncSDKManager` in your `AppDelegate` class in the same way you use it in Swift apps. Here's the pertinent template app code:

```
#import <SalesforceSDKCore/SalesforceSDKManager.h>
...
```

```
#import <MobileSync/MobileSyncSDKManager.h>
...

@implementation AppDelegate

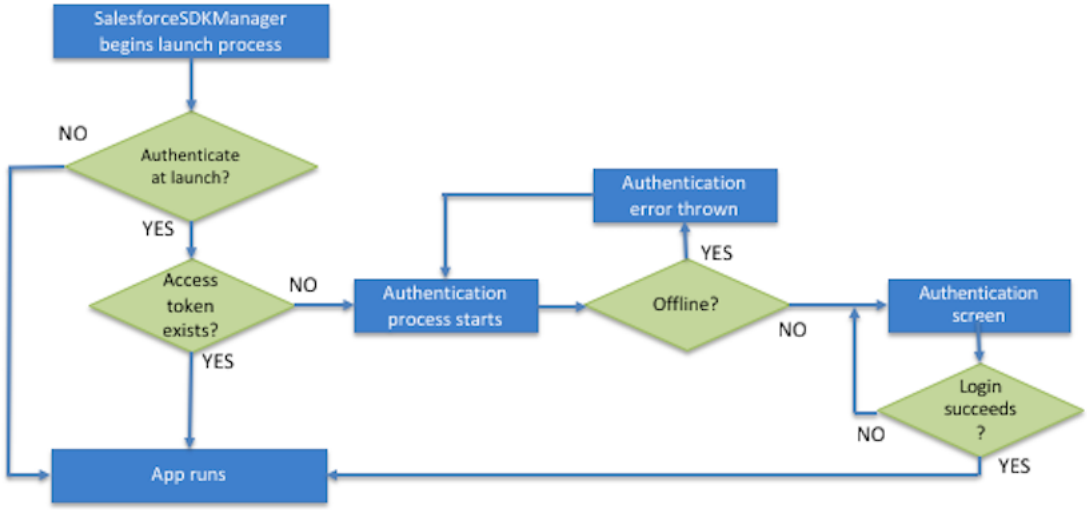
@synthesize window = _window;

- (instancetype)init
{
    self = [super init];
    if (self) {
        [MobileSyncSDKManager initializeSDK];
...
```

The `[MobileSyncSDKManager initializeSDK]` message ensures that the correct pieces are in place for using SmartStore, Mobile Sync, and everything in the core SDK. To use a different SDK manager object, import its header file and replace `MobileSyncSDKManager` in your code with the name of the appropriate manager class.

Application Launch Flow

When the `application(_:didFinishLaunchingWithOptions:)` message arrives, you initialize the app window and set your root view controller. If appropriate, Mobile SDK executes its login and authentication flow. If the app's connected app requires a passcode, the passcode verification screen appears before the bootstrap process can continue. The following diagram shows this flow.



Key points:

- If a valid access token is found, the flow bypasses Salesforce authentication.
- If no access token is found and the device is offline, the authentication module throws an error and returns the user to the login screen. `SalesforceSDKManager` doesn't reflect this event to the app.

Besides what's shown in the diagram, the `SalesforceSDKManager` launch process also delegates identity provider and push notification setup to apps that support those features. If the user fails or cancels either the passcode challenge or Salesforce authentication, an internal event fires. Control then returns to `AppDelegate`.

The `SalesforceSDKManager` object doesn't reappear until a user logout, user switch, or token expiration occurs.

Auth Helper Object

To handle login and logout events, Mobile SDK apps defer to the auth helper object.

Swift	Objective-C
`AuthHelper`	`SFSDKAuthHelper`

The auth helper class defines only type-level methods. After you initialize your app window, you can call `loginIfRequired` on the auth helper object. This call causes Mobile SDK to check the status of the most recent user. If Mobile SDK detects invalid OAuth tokens, it reruns the application launch flow.

To avoid unwanted loss of data or state during reinitialization, use auth helper handler methods. These methods let you define completion blocks (or handlers) for various logout scenarios. You can define one general logout handler or separate handlers for specific operations.

General Logout

```
handleLogout(completionBlock: (() → Void)!)
```

For performing operations that apply to all logout scenarios. Use only this method if your app doesn't support user switching.

Change Current User

```
registerBlock(forCurrentUserChangeNotifications(completionBlock: (() → Void)!)
```

For using a single completion block for logout and user switching operations. Calls the Logout Only method and then the Switch User method, passing your completion block to both methods.

Logout Only

```
registerBlock(forLogoutNotifications(completionBlock: (() → Void)!)
```

For handling logout operations in user-switching scenarios. Forwards your completion block to the `handleLogout` notification.

Switch User

```
registerBlock(forSwitchUserNotifications(completionBlock: (() → Void)!)
```

For handling user switching and initialization after a logout occurs.

See Also

`SFUserAccountManager`

AppDelegate Class

The `AppDelegate` class is the true entry point for an iOS app. In Mobile SDK template apps, `AppDelegate` implements a portion of the iOS `UIApplicationDelegate` interface. In a Mobile SDK Swift app with SwiftUI, `AppDelegate` performs these tasks:

- Initializes Mobile SDK through the `MobileSyncSDKManager` object
- Hands off control to `SceneDelegate` for Salesforce login and root view instantiation
- Optionally, registers your app for push notifications and sets up identity provider (IDP) functionality

Initialization

In the `init()` method of `AppDelegate`, the `SalesforceManager` object initializes Mobile SDK. The following code shows the `init` method as implemented by the template app.

```
override
init()
{
    super.init()
    MobileSyncSDKManager.initializeSDK
}
```

After `init()` returns, iOS calls `application(_:didFinishLaunchingWithOptions:_:)`. This call simply returns unless you've uncommented the line in it that registers your app for push notifications.

iOS next calls `application(_:configurationForConnecting:_:)` on `AppDelegate`. This method passes control to the `SceneDelegate` instance running on the UI thread. While `SceneDelegate` is launching the app's UI, `AppDelegate` continues to call push notification protocol methods. These notifications culminate in a prompt that asks the customer to allow or deny notifications.

If the app is currently active in the background, on foregrounding iOS performs an abbreviated version of this initialization process.

Initialization (Objective-C)

The following listing shows the `init` method as implemented by the template app. It is followed by a call to the `loginIfRequired:` method of `SalesforceSDKManager` in the `application:didFinishLaunchingWithOptions:` method.

```
- (instancetype)init
{
    self = [super init];
    if (self) {
        [MobileSyncSDKManager initializeSDK];

        //App Setup for any changes to the current authenticated user

        [SFSDKAuthHelper registerBlockForCurrentUserChangeNotifications:^{
            [self resetViewState:^{
                [self setupRootViewController];
            }];
        }];
    }
    return self;
}
```

In the `init` method, the `SalesforceSDKManager` object

- Initializes Mobile SDK.
- Registers a block to handle user change notifications. This block configures the app to call your `setupRootViewController` method to reset the view for the new user.

In your `AppDelegate` class, implement `setupRootViewController` to display your app's first screen after authentication. The `self.window` object must have a valid `rootViewController` by the time `application:didFinishLaunchingWithOptions:` completes. Here's the Mobile SDK Objective-C template's implementation.

```
- (void)setupRootViewController
{
    RootViewController *rootVC =
        [[RootViewController alloc] initWithNibName:nil bundle:nil];
    UINavigationController *navVC =
        [[UINavigationController alloc] initWithRootViewController:rootVC];
    self.window.rootViewController = navVC;
}
```

UIApplication Event Handlers

You can also use the `UIApplicationDelegate` protocol to implement `UIApplication` event handlers. Important event handlers that you might consider implementing or customizing include:

- **application(_:didFinishLaunchingWithOptions:)**
  Called at the beginning of your app's life-cycle. The template app implementation simply returns. Optionally, you can use this method to register the app for push notifications.
- **applicationDidBecomeActive(_:)**
  Called after the application is foregrounded.
- **application(_:didRegisterForRemoteNotificationsWithDeviceToken:)**
  Used for handling incoming push notifications from Salesforce. Follow the commented instructions and code in the template app's stub implementations.

For a list of all `UIApplication` event handlers, see the UIApplicationDelegate documentation.

SEE ALSO:

Using Push Notifications in iOS

SceneDelegate Class

The `SceneDelegate` class handles setup and life-cycle management for scenes. In the Mobile SDK Swift template, `SceneDelegate` also handles Salesforce logins and sets up your app's root view controller. Mobile SDK currently doesn't use `SceneDelegate` in its Objective-C template.

OAuth resources reside in an independent module. This separation makes it possible for you to use Salesforce authentication on demand. You can start the login process from within your `SceneDelegate` implementation, or you can defer it until it's actually required—for example, you can call OAuth from a scustom subview.

Initialization

In the following code, you can see how the template app sets up a handler for current user change event notifications.

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession,
    options connectionOptions: UIScene.ConnectionOptions) {

    guard let windowScene = (scene as? UIWindowScene) else { return }
    self.window = UIWindow(frame: windowScene.coordinateSpace.bounds)
    self.window?.windowScene = windowScene

    AuthHelper.registerBlock(forCurrentUserChangeNotifications: {
       self.resetViewState {
           self.setupRootViewController()
       }
    })
}
```

`AuthHelper`, a Mobile SDK shared object, registers the enclosed block to handle user change events. When this type of change occurs, Mobile SDK discards the former user's data. Here, the registered handler block calls `resetViewState()` to unwind any existing view hierarchy. Finally, `resetViewState()` calls `setupRootViewController()` to restore the app's original state.

```
func setupRootViewController() {
    // Setup store based on config userstore.json
    MobileSyncSDKManager.shared.setupUserStoreFromDefaultConfig()
    // Setup syncs based on config usersyncs.json
    MobileSyncSDKManager.shared.setupUserSyncsFromDefaultConfig()

    self.window?.rootViewController = UIHostingController(
        rootView: AccountsListView()
    )
}
```

Here, the template takes advantage of this execution point to set up SmartStore and Mobile Sync configurations—a one-time event per user. It then sets `AccountsListView` as the root view and creates a `UIHostingController` object based on it. This new object becomes the root view controller. Whenever your app enters the foreground—when it first loads or when a customer restores it from the background—, iOS calls `sceneWillEnterForeground(_:)`:

```
func sceneWillEnterForeground(_ scene: UIScene) {
    // Called as the scene transitions from the background to the foreground.
    // Use this method to undo the changes made on entering the background.
    self.initializeAppViewState()
    AuthHelper.loginIfRequired {
        self.setupRootViewController()
    }
}
```

The `initializeAppViewState()` method initializes or resets the state of your app's first screen. Before the call to `setupRootViewcontroller()`, you can customize the look and functionality of the login page's navigation bar. For example, create a method that customizes the public properties of the `SalesforceLoginViewControllerConfig` class.

```
func customizeLoginView() {
    let loginViewConfig = SalesforceLoginViewControllerConfig()

    // Set showSettingsIcon to false if you want to hide the settings
    // icon on the nav bar
    loginViewConfig.showsSettingsIcon = false

    // Set showNavBar to false if you want to hide the top bar
    loginViewConfig.showsNavigationBar = true
    loginViewConfig.navigationBarColor = UIColor(red: 0.051, green: 0.765, blue: 0.733,
alpha: 1.0)
    loginViewConfig.navigationTitleColor = UIColor.white
    loginViewConfig.navigationBarFont = UIFont(name: "Helvetica", size: 16.0)
    UserAccountManager.shared.loginViewControllerConfig = loginViewConfig
}
```

The `loginIfRequired` method takes a block argument that returns void and doesn't have parameters. Use this block, as shown, to set up your root view controller. Here, the `setupRootViewController` method sets the `rootViewController` property of `self.window` to your app's first custom view, thus launching your app's custom behavior. In this template, that view is the Accounts list view.

About Deferred Login

Beginning in Mobile SDK 7.0, you let Mobile SDK decide when it's necessary to show the login dialog. By calling `AuthHelper.loginIfRequired`, you nudge the SDK to consider whether the user has valid OAuth tokens. If not, Mobile SDK takes your advice and prompts the user to log in.

You're not required to call this method in your `SceneDelegate` implementation. You can defer login to any point after app initialization is completed. To defer authentication:

1. In the Xcode editor, open the `SceneDelegate` class.
2. In `sceneWillEnterForeground(_:)` method, remove the `loginIfRequired` call, but *keep the code that's inside the block*:

```
//AuthHelper.loginIfRequired {
    self.setupRootViewController() // Keep this call in sceneWillEnterForeground(_:)
//}
```

3. Call `AuthHelper.loginIfRequired { }` at the desired point of deferred login. To ensure that operations occur in the proper order, put the code that requires authentication in the closure parameter of `loginIfRequired:`.

RootViewController Class

A root view controller displays the first custom view that your app presents. In Objective-C apps, this view is the one at the bottom, or root, of your view stack. For the Mobile SDK Objective-C template project, Mobile SDK appropriately names its root view controller class `RootViewController`. This class sets up a mechanism for your app's interactions with the Salesforce REST API. Regardless of how you define your root view controller, you can reuse the template app's code for retrieving Salesforce data through REST APIs.

RootViewController Design

As the sole custom view controller in a basic Mobile SDK app, the `RootViewController` class covers only the bare essentials. Its two primary tasks are:

- Use Salesforce REST APIs to query Salesforce data
- Display the Salesforce data in a table

This app retrieves Salesforce data by issuing an asynchronous REST request in the form of a SOQL query. In this case, the query is a simple SELECT statement that gets the `Name` field from up to 10 Contact records. The app displays the query results in a static read-only table. Mobile SDK leverages the current user's authenticated credentials to form and send the REST request.

Swift Implementation

Beginning in Mobile SDK 9.0, the Swift template app no longer defines a `RootViewController` class. Instead, it uses SwiftUI views and models with Combine extensions to accomplish the same goals, as described in Native Swift Template.

Objective-C Implementation

In Objective-C, the `RootViewController` class inherits `UITableViewController` and implements the `SFRestDelegate` protocol. The action begins with an override of the `viewDidLoad` method of `UIViewController`:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
```

```
    self.title = @"Mobile SDK Sample App";

    SFRestRequest *request =
        [[SFRestAPI sharedInstance]
            requestForQuery:@"SELECT Name FROM Contact LIMIT 10"
                 apiVersion:kSFRestDefaultAPIVersion];
    [[SFRestAPI sharedInstance] send:request delegate:self];
}
```

After calling the superclass method and setting the view's title, the app creates and sends its REST request. Notice that the `requestForQuery` and `send:delegate:` messages are sent to a singleton shared instance of the `SFRestAPI` class. You use this singleton object for all REST requests.

After the apps sends the REST request, Salesforce responds by passing status messages and, hopefully, data to the delegate listed in the `send:delegate:` message. In this case, the delegate is the `RootViewController` object itself:

```
[[SFRestAPI sharedInstance] send:request delegate:self];
```

For handling REST responses, the Objective-C `RootViewController` uses the `SFRestDelegate` protocol instead of the `sendRestRequest:failBlock:completeBlock:` block method. The response arrives in one of the `SFRestDelegate` callback methods. For successful requests, the code handles Salesforce data in the `request:didSucceed:rawResponse:` callback:

```
- (void)request:(SFRestRequest *)request
     didSucceed:(id)jsonResponse
    rawResponse:(NSURLResponse *)rawResponse {

    NSArray *records = jsonResponse[@"records"];
    [SFLogger d:[self class]
         format:@"request:didLoadResponse: #records: %lu",
             (unsigned long)records.count];
    self.dataRows = records;
    dispatch_async(dispatch_get_main_queue(), ^{
        [self.tableView reloadData];
    });
}
```

As the use of the `id` data type suggests, this code handles JSON responses in generic Objective-C terms. It addresses the `jsonResponse` object as an instance of `NSDictionary` and treats its records as an `NSArray` object. Because `RootViewController` implements `UITableViewController`, it's simple to populate the table in the view with extracted records.

Native Swift Template

The native iOS Swift template supports the latest Swift features:

- SwiftUI
- Combine `Publisher`
- `SceneUI` and `SceneDelegate` flow

Also, this template demonstrates Mobile SDK offline features.

This template defines two initialization classes: `AppDelegate` and `SceneDelegate`. These classes interact to enable a dynamic arrangement of multiple windows.

Architecturally, this template uses strict model-view separation, with one model class per view class. Model classes implement the `ObservableObject` protocol, while view classes implement the `View` protocol.

AppDelegate

`AppDelegate` manages high-level setup—app launch and the `UISceneSession` lifecycle. In Mobile SDK terms, `AppDelegate` in this template

- Initializes Mobile SDK
- Registers and unregisters push notification services, if implemented
- Sets up IDP, if implemented

As in earlier Mobile SDK templates, you can customize settings in the `application(_:didFinishLaunchingWithOptions:)` method of `AppDelegate`.

SceneDelegate

`SceneDelegate` handles the life cycles of scenes in the `UISceneSession` container. The following initializer method configures a scene and adds it to the scene collection.

```
func scene(_:UIScene, willConnectTo session: UISceneSession, options connectionOptions:
UIScene.ConnectionOptions)
```

After instantiating the scene as a `UIWindow`, this method registers a handler for current user changes. If the `CurrentUserChange` event occurs, the template resets the view state by setting up the root view controller.

```
AuthHelper.registerBlock(forCurrentUserChangeNotifications: {
    self.resetViewState {
                self.setupRootViewController()
    }
})
```

Later, in the `sceneDidBecomeActive(_:)` method, the template reinitializes the app view state and resets the root view controller. If no user is logged into Salesforce, the template requires the user to log in before it performs this reset.

```
func sceneWillEnterForeground(_ scene: UIScene) {
    // Called as the scene transitions from the background to the foreground.
    //  Use this method to undo the changes made on entering the background.
    self.initializeAppViewState()
    AuthHelper.loginIfRequired {
        self.setupRootViewController()
    }
}
```

Model Classes

Models for the template's views differ in how they download Salesforce information.

- The `AccountsListModel` class uses the Mobile Sync `SyncManager` publisher to sync down information for all accounts. When the response arrives, the publisher stores the information in a SmartStore soup, according to the sync and SmartStore configurations.

  ```
  func fetchAccounts(){
      _ = syncManager?.publisher(for: "syncDownAccounts")
  ```

```
        .receive(on: RunLoop.main)
        .sink(receiveCompletion: { _ in }, receiveValue: { _ in
            self.loadFromSmartStore()
        })

    self.loadFromSmartStore()
}
```

To send account information to the view, the `loadFromSmartStore()` method uses the SmartStore publisher to extract the data from the SmartStore soup.

```
private func loadFromSmartStore(){
    storeTaskCancellable =
        self.store?.publisher(for: "select {Account:Name},
        {Account:Industry}, {Account:Id} from {Account}")
        .receive(on: RunLoop.main)
        .tryMap{
            $0.map { (row) -> Account in
                let r = row as! [String?]

                return Account(id: r[2] ?? "", name: r[0] ?? "",
                    industry: r[1] ?? "Unknown Industry" )
        }
    }
    .catch { error -> Just<[Account]> in
        print(error)
        return Just([Account]())
    }
    .assign(to: \AccountsListModel.accounts, on:self)
}
```

- The `ContactsForAccountModel` class uses a `RestClient` factory method to create the REST request.

```
let request = RestClient.shared.request(forQuery: "SELECT id, firstName, lastName, phone,
 email, mailingStreet, mailingCity, mailingState, mailingPostalCode FROM Contact WHERE
 AccountID = '\(acct.id)'", apiVersion: nil)
```

To send a request to Salesforce and process the asynchronous response, this model uses the `RestClient` shared Combine publisher.

```
contactsCancellable = RestClient.shared.publisher(for: request)
    .receive(on: RunLoop.main)
    .tryMap({ (response) -> Data in
        response.asData()
    })
    .decode(type: ContactResponse.self, decoder: JSONDecoder())
    .map({ (record) -> [Contact] in
        record.records
    })
    .catch( { error in
        Just([])
    })
    .assign(to: \.contacts, on:self)
...
```

This model stores details for all displayed contacts in a published array. See Handling REST Requests.

View Classes

Views include a list of accounts, and list and detail views of related contacts. The Account list view displays a list of account names. Selecting an account brings up a list view of the account's contacts. When the customer selects a contact record, a detail view displays the contact's information.

- **Accounts**
  - **Account** > **Contacts**
    - **Contact** > **Details**

Views serve to configure a visual interface that imports and presents data from the related model object. To handle taps on view items, the view object sets up a navigation link to the destination view and passes it the necessary data. Here's the definition for the Accounts list view.

```
struct AccountsListView: View {
  @ObservedObject var viewModel = AccountsListModel()

  var body: some View {
    NavigationView {
      List(viewModel.accounts) { dataItem in
        NavigationLink(destination:
            ContactsForAccountListView(account: dataItem)){
          HStack(spacing: 10) {
            VStack(alignment: .leading, spacing: 3) {
              Text(dataItem.name)
              Text(dataItem.industry).font(.subheadline).italic()
            }
          }
        }
      }
      .navigationBarTitle(Text("Accounts"), displayMode: .inline)
    }
    .onAppear{ self.viewModel.fetchAccounts() }
  }
}
```

See Also

- Scenes (Apple Developer Documentation)
- Ray Wenderlich's MVVM with Combine Tutorial for iOS
- https://www.vadimbulavin.com/ios-13-ipados-app-life-cycle-with-uiscene-scene-session-and-scene-delegate/

Using Salesforce REST APIs with Mobile SDK

The Salesforce API provides services for accessing Salesforce objects through REST endpoints. Mobile SDK supports these services by providing REST classes that

- Implement factory methods that create REST request objects for you.
- Send authenticated requests, based on your configuration, to Salesforce.
- Intercept the server's response and return it to your app as a Swift or Objective-C object.

In Mobile SDK for iOS, all REST requests are performed asynchronously. Responses for successful REST requests arrive in your app as `Array` or `Dictionary` objects. If a request fails, the response contains an `Error` object.

See Also

- Supported REST Services
- For an overview of all Salesforce APIs, see *Which API Do I Use?* in Salesforce Help.
- For information on Salesforce API request and response formats, see *REST API*.

Mobile SDK REST Client Interface

Swift	Objective-C
RestClient	SFRestAPI

This class defines the native interface for creating, formatting, and sending REST requests to the Salesforce service. When the service responds, this class relays the asynchronous response to either your implementation of the `RestClientDelegate` protocol or a callback block or closure that you define.

`RestClient` serves as a factory for `RestRequest` instances. It defines a request factory method for each supported Lightning Platform endpoint. Each factory method returns a `RestRequest` instance that is customized with your request parameter values.

`SFRestAPI` defines REST request factory methods in Objective-C without Swift renaming. To call these methods on the Swift `RestClient` object, use the autocomplete suggestions offered by the Xcode compiler. For example, type `RestClient.shared.request` in a Mobile SDK app, then choose from the list.

Warning: Because the Swift compiler determines method and parameter names heuristically, signatures can differ from their Objective-C equivalents.

See Also

- For a list of supported Salesforce Platform APIs, see Supported REST Services
- Creating REST Requests

REST Request Class

Swift	Objective-C	
`RestRequest`	`SFRestRequest`	

The `RestRequest` interface provides an uncomplicated way to make network calls. This class is the container for your request metadata, but you rarely manipulate the request object manually. The request object itself formats its metadata for transmission over HTTP. To send your requests to the Salesforce server, the `RestRequest` object configures the network parameters and the REST API syntax. If the object was created for a Salesforce resource, the class knows how to calculate the endpoint. Otherwise, you configure a custom path.

`RestRequest`—not `RestClient`— provides methods for configuring a REST request that isn't explicitly supported by Mobile SDK. For example, to use the Connect REST API or an external service, you manually create and configure a `RestRequest` object.

Creating REST Requests

The Mobile SDK REST API supports many types of Salesforce REST requests. For the Lightning Platform API, `RestClient` defines factory methods that return `RestRequest` instances for every supported endpoint. You obtain a customized `RestRequest` object by calling the appropriate REST client factory method with your request parameters. Factory `RestRequest` objects are specialized for the indicated request type and configured with your input values.

Here's an overview of how you can create REST requests.

- For the Lightning Platform API, `RestClient` factory methods return preformatted `RestRequest` objects based on minimal data input.
- (Objective-C only) `SFRestAPI (Blocks)` category methods let you define a REST request and send it in a single call. Block arguments, instead of a REST delegate object, handle REST responses.
- `SFRestAPI (Files)` category methods create `RestRequest` instances that provide access to Salesforce file-based resources.
- `RestRequest` methods support custom configurations for calling non-Lightning Platform and external APIs.
- `SFRestAPI (QueryBuilder)` category methods save you from having to manually format your own queries or searches. These methods construct SOQL query and SOSL search strings based on your input.

See Also

- Using REST Request Methods
- Unauthenticated REST Requests
- SFRestAPI (Blocks) Category
- SFRestAPI (QueryBuilder) Category
- SFRestAPI (Files) Category

Handling REST Requests

At runtime, Mobile SDK creates a singleton instance of `RestClient`. You use this instance to obtain a `RestRequest` object and to send that object to the Salesforce server.

Although you can use `RestRequest` objects for many CRUD operations, let's look at the most common use case: SOQL queries. Most other types of requests use the same flow with different input values. By default, SOQL query requests return up to 2,000 records per batch. In Mobile SDK 9.1 and later, you can specify a `batchSize` argument to customize the maximum batch size. This argument accepts any integer value between 200 and 2,000. Specifying a batch size does not guarantee that the returned batch is the requested size.

To create and send a SOQL REST request to the Salesforce server:

1. To create a request object, call the `RestClient` factory method that takes a SOQL query.

   Swift

   ```
   let request =
       = RestClient.shared.request(forQuery: "SELECT Name FROM Contact LIMIT 10",
   apiVersion: SFRestDefaultAPIVersion, batchSize: 500)
   ```

   Objective-C

   ```
   SFRestRequest *request = [[SFRestAPI sharedInstance]
       requestForQuery:@"SELECT Name FROM Contact LIMIT 10"];
   ```

2. Send your new REST request object using the REST client instance.

Swift

Option 1

Use the `RestClient` shared `Combine` publisher. For example, using the following request:

```
let request = RestClient.shared.request(forQuery: "SELECT id, firstName,
    lastName, phone, email, mailingStreet, mailingCity, mailingState,
    mailingPostalCode FROM Contact WHERE AccountID = '\(acct.id)'",
    apiVersion: nil, batchSize: 500)
```

Pass the request to the `RestClient.shared.publisher`. In case you've never encountered call chaining, the following code comments describe what happens in a typical publisher sequence. This code is from the `fetchContactsForAccount` method of the Swift template app's `ContactsForAccountModel` class.

```
// Use the RestClient publisher to send the request and return the raw REST response
contactsCancellable = RestClient.shared.publisher(for: request)
    // Receive the raw REST response object
    .receive(on: RunLoop.main)

    // Try to convert the raw REST response to Data
    // Throw an error in the event of failure
    .tryMap({ (response) -> Data in
        // Return the response as a Data byte buffer
        response.asData()
    })

    // Decode the Data object as JSON to produce an array
    // of Contacts formatted as a ContactResponse object
    .decode(type: ContactResponse.self, decoder: JSONDecoder())

    // Map the JSON array of Contacts
    .map({ (record) -> [Contact] in
        // Copy the Contact array to the records
        // member of ContactResponse
        record.records
    })

    // If tryMap failed, check for errors on the most recent
    // object in the chain
    .catch( { error in
        Just([])
    })

    // Store the array of contacts in the model's published contacts
    // property for use by ContactsForAccountListView
    .assign(to: \.contacts, on:self)
```

`Contact` and `ContactResponse` are structs that represent, respectively, a Contact record, and the formatted REST response to the SOQL query. Both objects are declared "Decodable".

```
struct Contact :  Identifiable, Decodable  {
    let id: UUID = UUID()
    let Id: String
    let FirstName: String?
```

```
    let LastName: String?
    let PhoneNumber: String?
    let Email: String?
    let MailingStreet: String?
    let MailingCity: String?
    let MailingState: String?
    let MailingPostalCode: String?
}

struct ContactResponse: Decodable {
    var totalSize: Int
    var done: Bool
    var records: [Contact]
}
```

Do you see the beauty of using the `RestClient` publisher? You can send the request and handle the response in a single block of logic.

Option 2

Pass the request object to the `send(request:_:)` method. Typically, Swift apps handle the asynchronous response in the trailing closure, as shown here.

```
RestClient.shared.send(request: request) { [weak self] (result) in
    switch result {
        case .success(let response):
            self?.handleSuccess(response: response, request: request)
        case .failure(let error):
            SalesforceLogger.d(RootViewController.self,
                message:"Error invoking: \(request) , \(error)")
    }
}
```

In the following alternate example, the calling object implements the `RestClientDelegate` protocol to handle responses. It therefore calls the `send(request:delegate:)` overload and passes itself as the delegate.

```
// This class implements the RestClientDelegate protocol to handle responses
RestClient.shared.send(request, delegate: self)
```

Objective-C

Pass the request object to the `send:` parameter of the `send:delegate:` method. Typically in Objective-C apps, the calling object handles the asynchronous response in its own implementation of the `SFRestDelegate` protocol, as shown here.

```
// This class implements the SFRestDelegate protocol
[[SFRestAPI sharedInstance] send:request delegate:self];
```

See Handling REST Responses.

Handling REST Responses

At the Salesforce server, all REST responses originate as JSON strings. Mobile SDK wraps this raw response in an iOS object and passes it to the app's handler. The response object's type follows the raw JSON structure:

- If the JSON root element is an object, the response is a Swift `Dictionary`.
- If the JSON root element is an array, the response is a Swift `Array`.

Your app is responsible for extracting the requested payload data from the response object. If you're not sure how the payload is structured, consult the REST API Developer Guide reference documentation.

Mobile SDK provides a variety of ways to handle REST responses, including:

- (*Swift only*) Use a `Combine` publisher. The `RestClient.shared.publisher` sends a SOQL request and receive the asynchronous response in the future. You can then chain calls to parse the raw response and format it for your app's use. See `ContactForAccountModel.swift` in the `iOSNativeSwiftTemplate` directory of the Templates repo.
- (*Swift only*) In a trailing closure after the `send(request:_:)` call. This approach is recommended for "vanilla" Mobile SDK apps in Swift.
- (*Swift, Objective-C*) By implementing the Mobile SDK REST delegate protocol and passing its ID to the REST client's `send` method. This approach is recommended for Objective-C apps.
- (*Objective-C only*) By implementing success and error blocks within the `send` method, using methods from the `SFRestApi (Blocks)` category.

Response Handling in Swift

Currently, the most modern and efficient approach is to use a `Combine` publisher, as explained in Handling REST Requests.

Using the REST Delegate Protocol

Swift	Objective-C
`RestClientDelegate`	`SFRestDelegate`

The REST delegate protocol is typically used in Objective-C apps, where it offer better code clarity than the `SFRestApi ( Blocks)` methods. Although the delegate approach is also available in Swift, you can achieve the same result more efficiently and clearly with publishers or trailing closures.

A class that adopts the REST delegate protocol becomes a potential target for asynchronous REST responses from Salesforce. When you send a REST request, you can tell the REST API client to forward the response to a specific delegate instance. Upon receipt, Mobile SDK routes that response to the appropriate method on the delegate instance you specified.

The `SFRestDelegate` protocol declares four possible responses:

- `request:didLoadResponse:`—Request was processed. The delegate receives the response in JSON format. This callback is the only one that indicates success.
- `request:didFailLoadWithError:`—Request couldn't be processed. The delegate receives an error message.
- `requestDidCancelLoad`—Request was canceled due to some external factor, such as administrator intervention, a network glitch, or another unexpected event. The delegate receives no return value.
- `requestDidTimeout`—The Salesforce server failed to respond in time. The delegate receives no return value.

The response arrives in a call to one of these delegate methods. Because you can't predict the type of response, you're required to implement all the methods. Responses in Swift and Objective-C delegates use the same method and parameter names.

Implementing REST Delegate Methods

Success Response

Swift	Objective-C
`request(request:didLoadResponse:[rawResponse:])`	`- (void)request:(SFRestRequest *)request didLoadResponse:(id)dataResponse rawResponse:(NSURLResponse *)rawResponse;`

The `request:didLoadResponse:` method is the only protocol method that handles a success condition, so place your code for handling Salesforce data in that method. For example:

```
- (void)request:(SFRestRequest *)request
        didLoadResponse:(id)jsonResponse {
    NSArray *records = [jsonResponse objectForKey:@"records"];
    NSLog(@"request:didLoadResponse: #records: %d", records.count);
    self.dataRows = records;
    [self.tableView reloadData];
}
```

If your method cannot infer the data type from the request, use introspection to determine the data type. For example:

```
if ([jsonResponse isKindOfClass:[NSArray class]]) {
     // Handle an NSArray here.
} else {
     // Handle an NSDictionary here.
}
```

You can address the response as an `NSDictionary` object and extract its records into an `NSArray` object. To do so, send the `NSDictionary:objectForKey:` message using the key "records".

Failed with Error Response

Swift	Objective-C
`request(request:didFailLoadWithError:)`	`request:didFailLoadWithError:`

A call to the `request:didFailLoadWithError:` callback results from one of the following conditions:

- If you use invalid request parameters, you get a `kSFRestErrorDomain` error code. For example, this error could indicate that you passed nil to `requestForQuery:`, or that you tried to update a non-existent object.
- If an OAuth access token expires, the framework tries to obtain a new access token and, if successful, retries the query. If a request for a new access token or session ID fails, you get a `kSFOAuthErrorDomain` error code. For example, the access token expires, and the OAuth refresh token is invalid. This scenario rarely occurs.
- If the low-level HTTP request fails, you get an `kSFIdentityErrorTypeBadHttpResponse` error code. For example, a Salesforce server becomes temporarily inaccessible.

requestDidCancelLoad and requestDidTimeout Methods

Swift	Objective-C
`requestDidCancelLoad(request:)`	`requestDidCancelLoad:`
`requestDidTimeout(request:)`	`requestDidTimeout:`

The "load canceled" and "request timed out" delegate methods are self-describing and don't return an error code. You can choose to handle the result however you want: display an error message, write to the log, retry the request, and so on.

Batch and Composite Requests

Batch and composite APIs pose special challenges, because they handle multiple requests in a single call. In Swift, Mobile SDK extensions reduce the pain of building and configuring these complex requests.

Batch and Composite Request Classes

Swift

- `BatchRequest`
- `BatchRequestBuilder`
- `BatchResponse`
- `CompositeRequest`
- `CompositeRequestBuilder`
- `CompositeResponse`

Objective-C

- `SFSDKBatchRequest`
- `SFSDKBatchRequestBuilder`
- `SFSDKBatchResponse`
- `SFSDKCompositeRequest`
- `SFSDKCompositeRequestBuilder`
- `SFSDKCompositeResponse`

These classes make it easy to create batch and composite requests. To use them:

1. Create a builder instance. For batch requests, you can optionally set `haltOnError` property to true:

   ```
   let builder = BatchRequestBuilder()
   // Optional; defaults to false
   builder.setHaltOnError(true)
   ```

   For composite requests, you can optionally set the `allOrNone` rollback property to true.

   ```
   let builder = CompositeRequestBuilder()
   // Optional; defaults to false
   builder.setAllOrNone(true)
   ```

2. As you create REST requests, add them to the builder object.

```
builder.add(request as! BatchRequest)
```

With composite requests, you also provide a reference ID as described in the *REST API Developer Guide*. You can add a base `RestRequest` object and specify the reference ID explicitly:

```
builder.add(request, referenceId)
```

or add a `CompositeSubRequest` object, which is a `RestRequest` object that stores the reference ID internally:

```
builder.add(request as! CompositeSubRequest)
```

3. When you're ready, call the builder object's build method—`buildBatchRequest(_:)` or `buildCompositeRequest(_:)`. For example:

```
let request = builder.buildBatchRequest(SFRestDefaultAPIVersion)
```

Each build method returns a specialized `BatchRestRequest` or `CompositeRestRequest` object. You can send these objects through the shared `RestClient` instance.

Tip: To use an older API version as the default argument, pass a literal string in the format "v42.0".

4. Responses to batch and composite requests arrive as instances of the response class (`BatchResponse` or `CompositeResponse`).

See Also

- "Batch" in *REST API Developer Guide*
- "Composite" in *REST API Developer Guide*

Using REST Request Methods

Salesforce offers some product-specific REST APIs that have no relationship to Salesforce Platform APIs. You can use Mobile SDK resources to configure and send such requests. For these cases, you create and configure your `RestRequest` object directly, instead of relying on factory methods.

To send a non-SOQL and non-SOSL REST request using `RestClient`:

1. Create an instance of `RestRequest`.
2. Set the properties you need on the `RestRequest` object.
3. Call `send` on the `RestClient` object, passing in the `RestRequest` object you created as the first parameter.

The following example performs a GET operation to obtain all items in a specific Chatter feed.

```
let request = RestRequest(method: .GET, serviceHostType: .instance,
    path: "/v42.0/chatter/feeds/user-profile/me/feed-elements",
    queryParams: nil)
RestClient.shared.send(request: request { [weak self] (result) in
    switch result {
        case .success(let response):
            // do something with response
        case .failure(let error):
            SalesforceLogger.d(RootViewController.self, message:"Error invoking:
\(request) , \(error)")
```

```
    }
}
```

4. To perform a request with parameters, wrap your parameter string using `SFJsonUtils.object(fromJSONString:)`, and assign it to the `queryParams` property of `RestRequest`. To send a custom request, create a `Dictionary` object and use the `setCustomRequestBodyData(_:contentType:)` method of `RestRequest`.

   The following example uses a custom request body to add a comment to a Chatter feed.

```
let dataString = "{\"body\" : {\"messageSegments\" : [{\"type\" : \"Text\", " +
    "\"text\" : \"Some Comment\"}]}, \"feedElementType\":\"FeedItem\", " +
    "\"subjectId\":\"me\"}"

if let data = dataString.data(using: .utf8) {

    let request = RestRequest(method: .POST, serviceHostType: .instance,
        path: "/v42.0/chatter/feed-elements", queryParams: nil)
    request.setCustomRequestBodyData(data, contentType: "application/json")

    RestClient.shared.send(request: request { [weak self] (result) in
        switch result {
            case .success(let response):
                // do something with response
            case .failure(let error):
                SalesforceLogger.d(RootViewController.self, message:"Error invoking:
\(request) , \(error)")
        }
    }
}
```

5. To set an HTTP header for your request, use the `setHeaderValue(_:forHeaderName:)` method. This method can help you when you're displaying Chatter feeds, which come pre-encoded for HTML display. To avoid displaying unwanted escape sequences in Chatter comments, set the `X-Chatter-Entity-Encoding` header of your request to "false":

```
...
request.setHeaderValue("false", forHeaderName:"X-Chatter-Entity-Encoding")
```

Unauthenticated REST Requests

Most REST requests from Mobile SDK apps go to secure Salesforce endpoints on behalf of an authenticated Salesforce customer. For these cases, Mobile SDK handles authentication for the app by embedding the current user's OAuth token in the request and automatically refreshing stale tokens.

Sometimes, however, Mobile SDK apps request network services without authentication. For example:

- Before the user has logged in to Salesforce
- Through an unauthenticated endpoint within Salesforce
- Through an unauthenticated endpoint outside of Salesforce

Mobile SDK provides two versions of its REST client:

- `RestClient.shared`—initialized with the current user's credentials
- `RestClient.sharedGlobal`—not initialized with a user

You can access authenticated and unauthenticated endpoints with the `RestClient.shared` instance. However, you can access only unauthenticated endpoints with the `RestClient.sharedGlobal` instance. How you access unauthenticated endpoints depends on whether the service is in the Salesforce domain.

Note: Unauthenticated REST requests require a full path URL. Mobile SDK doesn't prepend an instance URL to unauthenticated endpoints.

Requesting Unauthenticated Salesforce Resources

An unauthenticated endpoint is one that doesn't require an OAuth token. Few Lightning Platform APIs are unauthenticated, but other products or your own Apex endpoints might be. To configure a request for an unauthenticated Salesforce resource, set its `requiresAuthentication` property to `false` or `NO`.

Swift

```
request.requiresAuthentication = false
```

Objective-C

```
request.requiresAuthentication = NO;
```

Requesting Unauthenticated External Resources

To support requests to unauthenticated external endpoints, Mobile SDK provides a shared global instance of its REST client. This REST client doesn't require OAuth authentication and is unaware of the concept of user. Native apps can use it to send custom unauthenticated requests to non-Salesforce endpoints before or after the user logs in to Salesforce.

Here's how you access the shared global REST client:

Swift:

```
RestClient.sharedGlobal
```

Objective-C:

```
[[SFRestAPI sharedGlobalInstance]
```

To call an external endpoint, use the `RestRequest.customUrlRequest(with:baseURL:path:queryParams:)` method.

Swift

```
let request =
    RestRequest.customUrlRequest(with: RestRequest.Method.GET,
                                 baseURL: "https://api.github.com",
                                    path: "/orgs/forcedotcom/repos",
                             queryParams: nil)
request.requiresAuthentication = false
RestClient.sharedGlobal.send(request: request,
    onFailure: {(error, rawResponse) in

}) {(response, rawResponse) in

}
```

Objective-C

```
SFRestRequest *request =
    [SFRestRequest customUrlRequestWithMethod:SFRestMethodGET
                                      baseURL:@"https://api.github.com"
                                         path:@"/orgs/forcedotcom/repos"
                                  queryParams:nil];
request.requiresAuthentication = NO;
[[SFRestAPI sharedGlobalInstance] sendRESTRequest:request
    failBlock:^(NSError *  e, NSURLResponse * rawResponse) {

    } completeBlock:^(id  resp, NSURLResponse *  rawResponse) {
        NSDictionary *response = resp;
        // response should have forcedotcom repos
}];
```

After the customer authenticates with Salesforce, the app can switch to the authenticated REST client. You can call any endpoint through the authenticated instance, but you cannot call authenticated endpoints through the global instance.

Requesting Unauthenticated Resources in Non-Native Apps

For non-native apps—hybrid and React Native—Mobile SDK does not support a REST client object. To access unauthenticated endpoints in non-native apps, call one of the following methods and pass false to its `doesNotRequireAuthentication` parameter:

Swift

```
function anyrest(fullUrlPath, returnsBinary, doesNotRequireAuthentication,
params, successHandler, errorHandler)
```

React Native

```
sendRequest(endPoint, path, successCB, errorCB, method, payload, headerParams,
fileParams, returnBinary, doesNotRequireAuthentication)
```

SFRestAPI (Blocks) Category

For receiving and handling REST API responses, you can use inline code blocks instead of a delegate class. This alternative Objective-C approach lets you send a request and handle its asynchronous response in a single method call.

Mobile SDK for native iOS provides block methods for single requests, composite requests, and batch requests. When you use these methods, you provide block arguments to handle success and failure responses. Mobile SDK forwards the asynchronous response to your success or failure block according to the response's network status.

Block methods and associated typedefs are defined in the `SFRestAPI (Blocks)` category as follows:

```
// Sends a single request you've already built, using blocks to return status.
- (void) sendRequest:(SFRestRequest *)request
        failureBlock:(SFRestRequestFailBlock)failureBlock
        successBlock:(SFRestResponseBlock)successBlock
        NS_REFINED_FOR_SWIFT;

// Sends a composite request you've already built, using blocks to return status.
- (void) sendCompositeRequest:(SFSDKCompositeRequest *)request
                 failureBlock:(SFRestRequestFailBlock)failureBlock
                 successBlock:(SFRestCompositeResponseBlock)successBlock
                 NS_REFINED_FOR_SWIFT;
```

```
// Sends a batch request you've already built, using blocks to return status.
- (void) sendBatchRequest:(SFSDKBatchRequest *)request
             failureBlock:(SFRestRequestFailBlock)failureBlock
             successBlock:(SFRestBatchResponseBlock)successBlock
             NS_REFINED_FOR_SWIFT;
```

Each `send` method requires two blocks:

Failure Block

A failure block can receive timeout, cancellation, or error failures.

- Block type:

```
typedef void (^SFRestRequestFailBlock) (id _Nullable response,
    NSError * _Nullable e, NSURLResponse * _Nullable rawResponse)
    NS_SWIFT_NAME(RestRequestFailBlock);
```

- Response type: `NSError`

Success Block

Block type and response type depend on the request type as follows:

Single Request

- Response type: `NSArray` or `NSData` object, depending on the data returned.
- Block types (use the appropriate template for your request's return type):

```
typedef void (^SFRestDictionaryResponseBlock) (NSDictionary * _Nullable dict,
    NSURLResponse * _Nullable rawResponse)
    NS_SWIFT_NAME(RestDictionaryResponseBlock);

// Use this block when you request API versions
typedef void (^SFRestArrayResponseBlock) (NSArray * _Nullable arr,
    NSURLResponse * _Nullable rawResponse)
    NS_SWIFT_NAME(RestArrayResponseBlock);
```

Composite Request

- Block type:

```
typedef void (^SFRestCompositeResponseBlock) (SFSDKCompositeResponse *response,
    NSURLResponse * _Nullable rawResponse)
    NS_SWIFT_NAME(RestCompositeResponseBlock);
```

- Response type: `SFSDKCompositeResponse`

Batch Request

- Block type:

```
typedef void (^SFRestBatchResponseBlock) (SFSDKBatchResponse *response,
    NSURLResponse * _Nullable rawResponse) NS_SWIFT_NAME(RestBatchResponseBlock);
```

- Response type: `SFSDKBatchResponse`

Send a Request Using Blocks

To send a request using a block method:

1. Create the `SFRestRequest` object that fits your needs.
   - For a single request:
     - Create your `SFRestRequest` object by calling the appropriate `SFRestAPI` factory method on page 527.
     - Send your request. The following example handles any request that returns an `NSDictionary` response:

```
- (void)sendRequestForDictionary:(SFRestRequest *)request {
    // Block to handle failure
   SFRestRequestFailBlock failBlock = ^(id response, NSError *error, NSURLResponse
 *rawResponse) {
        // Do as you wish with the error information
        //...
    };

    // Block to handle success
    SFRestDictionaryResponseBlock completeBlock = ^(NSDictionary *data,
NSURLResponse *rawResponse) {
        // Process the dictionary data
        //...
    };

    // Send the request to Salesforce along with your success and failure blocks
    [[SFRestAPI sharedGlobalInstance] sendRequest:request
                                        failureBlock:failBlock
                                     successBlock:completeBlock];
...
}
```

   - For batch and composite requests:
     a. For each subrequest, create an `SFRestRequest` object by calling an appropriate factory method.
     b. To create the `SFRestRequest` object that you send to Salesforce, pass an array of your subrequest objects to the batch or composite factory method. See SFRestAPI.h for more information.

```
// Batch request factory method
- (SFRestRequest *) batchRequest:(NSArray<SFRestRequest *> *)requests
                     haltOnError:(BOOL)haltOnError
                      apiVersion:(nullable NSString *)apiVersion;

// Composite request factory method
- (SFRestRequest *)compositeRequest:(NSArray<SFRestRequest *> *) requests
                            refIds:(NSArray<NSString *> *)refIds
                         allOrNone:(BOOL)allOrNone
                        apiVersion:(nullable NSString *)apiVersion;
```

2. Design your success block to handle the expected data.
3. Design your failure block to handle the issue gracefully.
4. Pass the `SFRestRequest` object you created in step 1, and your success and failure blocks, to the appropriate `send` block method.

Note:

- For Swift, Mobile SDK refines the Objective-C block methods to funnel the REST response into a single Swift completion closure. See Handling REST Responses on page 87.
- In Objective-C, judicious use of blocks and delegates can help fine-tune your app's readability and ease of maintenance. Ideal conditions for using blocks often correspond to those that mandate inline functions in C++ or anonymous functions in Java. Ultimately, you make a judgment call based on your own requirements.

SFRestAPI (QueryBuilder) Category

If you're unsure of the correct syntax for a SOQL query or a SOSL search, you can get help from the `SFRestAPI (QueryBuilder)` category methods. These methods build query strings from basic conditions that you specify, and return the formatted string. You can pass the returned string to `SFRestAPI` or `RestClient` request methods for query or search.

`SFRestAPI (QueryBuilder)` provides two static methods each for SOQL queries and SOSL searches: one takes minimal parameters, while the other accepts a full list of options. Swift versions of these methods are not defined explicitly by Mobile SDK. To code these methods in Swift, use the autocomplete suggestions offered by the Xcode compiler. These suggested method and parameter names are determined by Swift compiler heuristics and can differ from their Objective-C equivalents.

SOSL Methods

SOSL query builder methods are:

Swift (Compiler-generated)

```
RestClient.soslSearch(withSearchTerm:objectScope:)
RestClient.soslSearch(withSearchTerm:fieldScope:objectScope:limit:)
```

Objective-C

```
+ (NSString *) SOSLSearchWithSearchTerm:(NSString *)term
                            objectScope:(NSDictionary *)objectScope;


+ (NSString *) SOSLSearchWithSearchTerm:(NSString *)term
                             fieldScope:(NSString *)fieldScope
                            objectScope:(NSDictionary *)objectScope
                                  limit:(NSInteger)limit;
```

Parameters for the SOSL search methods are:

Swift	Objective-C
`withSearchTerm`	`term`

  The search string. This string can be any arbitrary value. The method escapes any SOSL reserved characters before processing the search.

Swift	Objective-C
`fieldScope`	`fieldScope`

Indicates which fields to search. It's either `nil` or one of the IN search group expressions: "IN ALL FIELDS", "IN EMAIL FIELDS", "IN NAME FIELDS", "IN PHONE FIELDS", or "IN SIDEBAR FIELDS". A `nil` value defaults to "IN NAME FIELDS". See Salesforce Object Search Language (SOSL).

Swift	Objective-C
`objectScope`	`objectScope`

  Specifies the objects to search. Acceptable values are:

  - `nil`—No scope restrictions. Searches all searchable objects.
  - An `NSDictionary` object pointer—Corresponds to the SOSL RETURNING fieldspec. Each key is an `sObject` name; each value is a string that contains a field list as well as optional WHERE, ORDER BY, and LIMIT clauses for the key object.

    If you use an `NSDictionary` object, each value must contain at least a field list. For example, to represent the following SOSL statement in a dictionary entry:

    ```
    FIND {Widget Smith}
    IN Name Fields
    RETURNING Widget__c (name Where createddate = THIS_FISCAL_QUARTER)
    ```

    set the key to "Widget__c" and its value to "name WHERE `createddate` = "THIS_FISCAL_QUARTER". For example:

    ```
    [SFRestAPI
        SOSLSearchWithSearchTerm:@"all of these will be escaped:~{]"
                    objectScope:[NSDictionary
                                 dictionaryWithObject:@"name WHERE
                                                       createddate="THIS_FISCAL_QUARTER"
                                                       forKey:@"Widget__c"]];
    ```

  - `NSNull`—No scope specified.

Swift	Objective-C
`limit`	`limit`

  To limit the number of results returned, set this parameter to the maximum number of results you want to receive.

SOQL Methods

SOQL QueryBuilder methods that construct SOQL strings are:

Swift (Compiler-generated)

```
RestClient.soqlQuery(withFields:sObject:whereClause:limit:)
RestClient.soqlQuery(withFields:sObject:whereClause:groupBy:having:orderBy:limit:)
```

Objective-C

```
+ (NSString *) SOQLQueryWithFields:(NSArray *)fields
                           sObject:(NSString *)sObject
                             where:(NSString *)where
                             limit:(NSInteger)limit;
```

```
+ (NSString *) SOQLQueryWithFields:(NSArray *)fields
                           sObject:(NSString *)sObject
                             where:(NSString *)where
                           groupBy:(NSArray *)groupBy
                            having:(NSString *)having
                           orderBy:(NSArray *)orderBy
                             limit:(NSInteger)limit;
```

Parameters for the SOQL methods correspond to SOQL query syntax. All parameters except `withFields/fields` and `sObject` can be set to `nil`.

Swift	Objective-C
`withFields`	`fields`

  An array of field names to be queried.

Swift	Objective-C
`sObject`	`sObject`

  Name of the object to query.

Swift	Objective-C
`whereClause`	`where`

  An expression specifying one or more query conditions.

Swift	Objective-C
`groupBy`	`groupBy`

  An array of field names to use for grouping the resulting records.

Swift	Objective-C
`having`	`having`

  An expression, usually using an aggregate function, for filtering the grouped results. Used only with `groupBy`.

Swift	Objective-C
`orderBy`	`orderBy`

  An array of fields name to use for ordering the resulting records.

Swift	Objective-C
`limit`	`limit`

  Maximum number of records you want returned.

See SOQL SELECT Syntax.

SOSL Sanitizing

The `QueryBuilder` category also provides a class method for cleaning SOSL search terms:

Swift

```
RestClient.sanitizeSOSLSearchTerm(_:)
```

Objective-C

```
+ (NSString *) sanitizeSOSLSearchTerm:(NSString *)searchTerm;
```

This method escapes every SOSL reserved character in the input string, and returns the escaped version. For example:

Swift

```
let cleanSOSL = RestClient.sanitizeSOSLSearchTerm(_: "FIND {MyProspect}")
```

Objective-C

```
NSString *soslClean = [SFRestAPI sanitizeSOSLSearchTerm:@"FIND {MyProspect}"];
```

This call returns "FIND \{MyProspect\}".

The `sanitizeSOSLSearchTerm` method is called in the implementation of the SOSL and SOQL QueryBuilder methods, so you don't need to call it on strings that you're passing to those methods. However, you can use it if, for instance, you're building your own queries manually. SOSL reserved characters include:

\ ? & | ! { } [ ] ( ) ^ ~ * : " ' + -

`SFRestAPI (Files)` Category

The `SFRestAPI (Files)` category provides methods that generate file operation requests. Each method returns a new `RestRequest` or `SFRestRequest` object. Applications send this object to the Salesforce service to process the request.

See Also

- Files and Networking.
- "Files Methods" at https://forcedotcom.github.io/SalesforceMobileSDK-iOS/Documentation/SalesforceSDKCore/html/Classes/SFRestAPI.html.
- For descriptions of Files REST resources, see **Connect REST API Resources** > **Files Resources** at http://www.salesforce.com/us/developer/docs/chatterapi.

Handling Authentication Errors

Swift	Objective-C
`UserAccountManager`	`SFUserAccountManager`
`UserAccountManagerDelegate`	`SFUserAccountManagerDelegate`

Mobile SDK provides default error handlers that display messages and divert the app flow when authentication errors occur. In an error event, the user account manager iterates through its delegates and gives them the chance to handle the error.

To insert your own classes into this chain, implement the delegate protocol and override the following method:

Swift:

```
func userAccountManager(_ accountManager: UserAccountManager,
                didFailAuthenticationWith: Error,
                                     info: AuthInfo) -> Bool
{
    // Provide custom error handling
    //...
    return true
}
```

Objective-C:

```
/**
 *
 * @param userAccountManager The instance of SFUserAccountManager
 * @param error The Error that occurred
 * @param info  The info for the auth request
 * @return YES if the error has been handled by the delegate.
 * SDK will attempt to handle the error if the result is NO.
 */
- (BOOL)userAccountManager:(SFUserAccountManager *)userAccountManager
                     error:(NSError*)error
                      info:(SFOAuthInfo *)info {
    // Provide custom error handling
    //...
    return YES;
}
```

A return value of `true` or `YES` indicates that the method handled the current error condition. In this case, the user account manager takes no further action for this error. Otherwise, the delegate did not handle the error, and the error handling process falls to the next delegate in the list. If all delegates return `false`, the user account manager uses its own error handler.

Note: For authentication error handling, Mobile SDK historically used a customizable list of `SFAuthErrorHandler` objects in the `SFAuthenticationManager` shared object. `SFAuthenticationManager` is now deprecated. If you had customized the authentication error handler list, update your code to use `UserAccountManagerDelegate` (Swift) or `SFUserAccountManagerDelegate` (Objective-C).

Supporting iPadOS in Mobile SDK Apps

Mobile SDK 9.0 enhances the iPad customer experience. Landscape mode now functions as expected, and Mobile SDK now supports multiple windows.

Swift	Objective-C
`AuthHelper`	SFSDKAuthHelper

While Mobile SDK apps for iPad aren't required to explicitly enable multiple windows, use of this feature does require changes to your `SceneDelegate` code. New methods on `AuthHelper` add a `UIScene` parameter to existing methods of the same name. To Mobile SDK, the scene clarifies the login flow, allowing SDK objects to adjust to the shifting customer focus. When a user logs in or logs out in one scene, the change applies to all scenes. You can handle each case after the fact by defining the `completion` callback blocks in the `loginIfRequired(_:completion:)` and `handleLogout(_:completion:)` methods.

To see the new methods in action, study the updated `SceneDelegate` class in the RestAPIExplorer sample app. The `registerBlock(forCurrentUserChangeNotifications:)` and `loginIfRequired(_:completion:)` methods of the `AuthHelper` class now accept a `UIWindowScene` argument.

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options
connectionOptions: UIScene.ConnectionOptions) {
    guard let windowScene = (scene as? UIWindowScene) else { return }
    self.window = UIWindow(frame: windowScene.coordinateSpace.bounds)
    self.window?.windowScene = windowScene

    AuthHelper.registerBlock(forCurrentUserChangeNotifications: scene) {
        self.resetViewState {
            self.setupRootViewController()
        }
    }
    self.initializeAppViewState()

    AuthHelper.loginIfRequired(scene) {
        self.setupRootViewController()
    }
}
```

In the `completion` block of `loginIfRequired(_:completion:)`, this sample app calls code that resets the scene to its beginning state—in this case, an instance of `RootViewController`. The `registerBlock(forCurrentUserChangeNotifications:completion:)` completion block first discards the view stack of the outgoing user, then resets the scene to `RootViewController`. To support multiple windows, copy these two calls into the `scene(_:willConnectTo:_:)` method of your Swift app's `SceneDelegate` instance. Replace the existing `AuthHelper.registerBlock(_:)` call with the scene-enabled version.

IDP Apps

Similarly to `AppDelegate`, `SceneDelegate` provides a method for opening URLs. To support multiple windows, Mobile SDK IDP client apps must use the scene delegate and pass in the scene's persistent identifier.

```
func scene(_ scene: UIScene, openURLContexts URLContexts:
```

```
    Set<UIOpenURLContext>) {

    if let urlContext = URLContexts.first {
        UserAccountManager.shared.handleIdentityProviderResponse(
            from: urlContext.url,
            with: [UserAccountManager.IDPSceneKey:
                scene.session.persistentIdentifier])
    }
}
```

AuthHelper Scene-Enabled Methods

The following `AuthHelper` methods are new overloads that add a `UIScene` parameter to existing methods.

Swift

```
func loginIfRequired(_ scene: UIScene, completion completionBlock: (() -> Void)?)

func handleLogout(_ scene: UIScene, completion completionBlock: (() → Void)?)

func registerBlock(forCurrentUserChangeNotifications scene: UIScene completion
completionBlock: () → Void)

func registerBlock(forLogoutNotifications scene:UIScene, completion completionBlock: ()
 → Void)
```

Objective-C

```
+ (void)loginIfRequired:(UIScene *)scene completion:
    (nullable void (^)(void))completionBlock;

+ (void)handleLogout:(UIScene *)scene completion:
    (nullable void (^)(void))completionBlock;

+ (void)registerBlockForCurrentUserChangeNotifications:
    (UIScene *)scene completion:(void (^)(void))completionBlock;

+ (void)registerBlockForLogoutNotifications:
    (UIScene *)scene completion:(void (^)(void))completionBlock;
```

Supporting Catalyst in Mobile SDK Apps

Beginning in version 9.1, Mobile SDK adds Catalyst support to its native frameworks. Catalyst is an Apple cross-platform product that enables iPad apps to run on Intel and Silicon Macs. Preparing Mobile SDK iPad apps to run on macOS is easy—it usually requires only a couple of additional settings in your iOS project configuration.

Mobile SDK supports macOS through Catalyst in the following frameworks:

- SalesforceSDKCommon
- SalesforceAnalytics
- SalesforceSDKCore
- SmartStore

- MobileSync

The following Mobile SDK frameworks don't support Catalyst:

- SalesforceHybridSDK
- SalesforceReact
- SalesforceFileLogger framework (currently in development)

Note: To run on macOS, all third-party frameworks that your app uses must also support Catalyst.

Get Started with Catalyst

To configure your iPad apps for macOS, follow Apple's Turning on Mac Catalyst tutorial. In the Mac version of your app, be sure to set the deployment target to **macOS 11**.

Mobile SDK also provides examples that demonstrate Catalyst support:

MobileSyncExplorerSwift Template App
You can use this template with the `forceios createwithtemplate` command to create a Catalyst-ready project.

RestAPIExplorer Sample App

Special Considerations for Mobile SDK Security Features

Mobile SDK apps that use the following security features require special consideration for Catalyst.

Passcode
Passcode is a useful security gate for mobile devices such as iPhones and iPads. But does a passcode make sense when your app's running on a Mac? We leave the decision to you—Mobile SDK passcode support is still available and works as expected on Macs.

Snapshot
Snapshot is essential on mobile devices for reasons that don't apply to Macs. Mobile SDK automatically disables this feature in Mac versions of your app.

Using iOS App Extensions with Mobile SDK

iOS app extensions provide opportunities for developers to extend their app's functionality beyond the app window. Mobile SDK supports app extensions with only a small amount of extra configuration.

About iOS App Extensions in Mobile SDK Apps

An iOS app extension is a separate target in your project. It lives in its own folder and is added to your app bundle as a separate binary unit. However, the app extension can access the same resources and libraries as your main app. Apple offers many extension types, as described in App Extensions articles at developer.apple.com.

To enable extensions, you add special configuration in two areas:

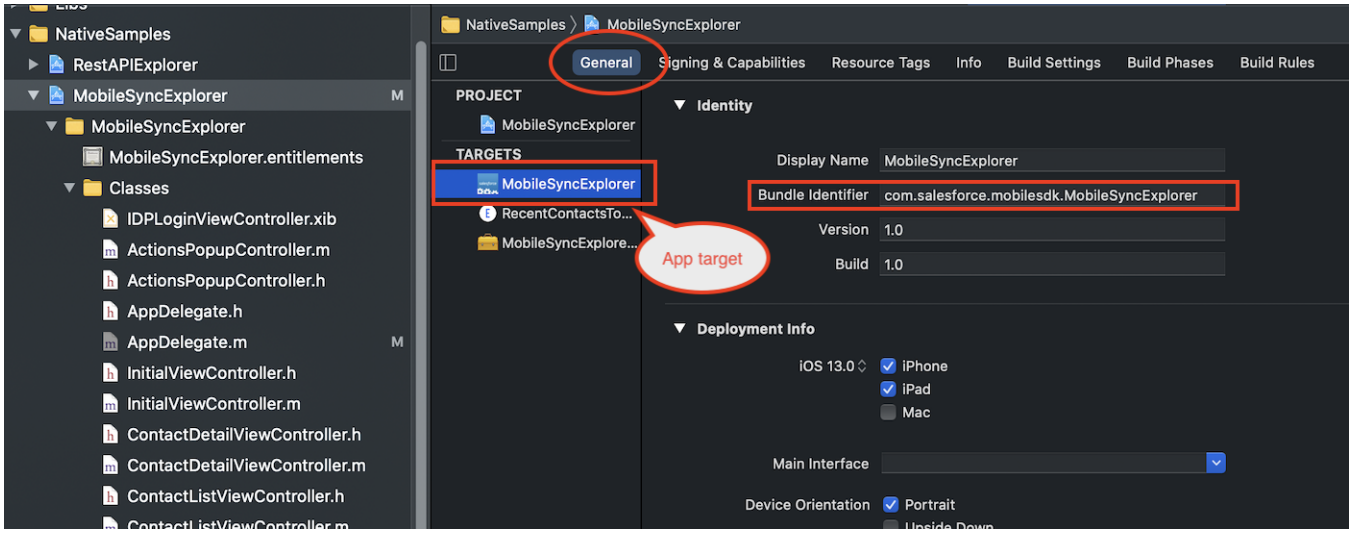
- **Project settings:** When you add an app extension, Xcode creates an app extension build target. You configure the existing main target and the new extension target to join the same app group and keychain sharing group. Both apps must sign in to the same provisioning profile.
- **Application code:** At runtime, the two apps must use identical app group information, and the main app must convey user authentication status to the extension.

After everything is properly configured, your app extension can run any Mobile SDK code that's appropriate for the extension type.

Note: You can reliably test app extensions only on physical devices. Simulator tests can give unpredictable results.

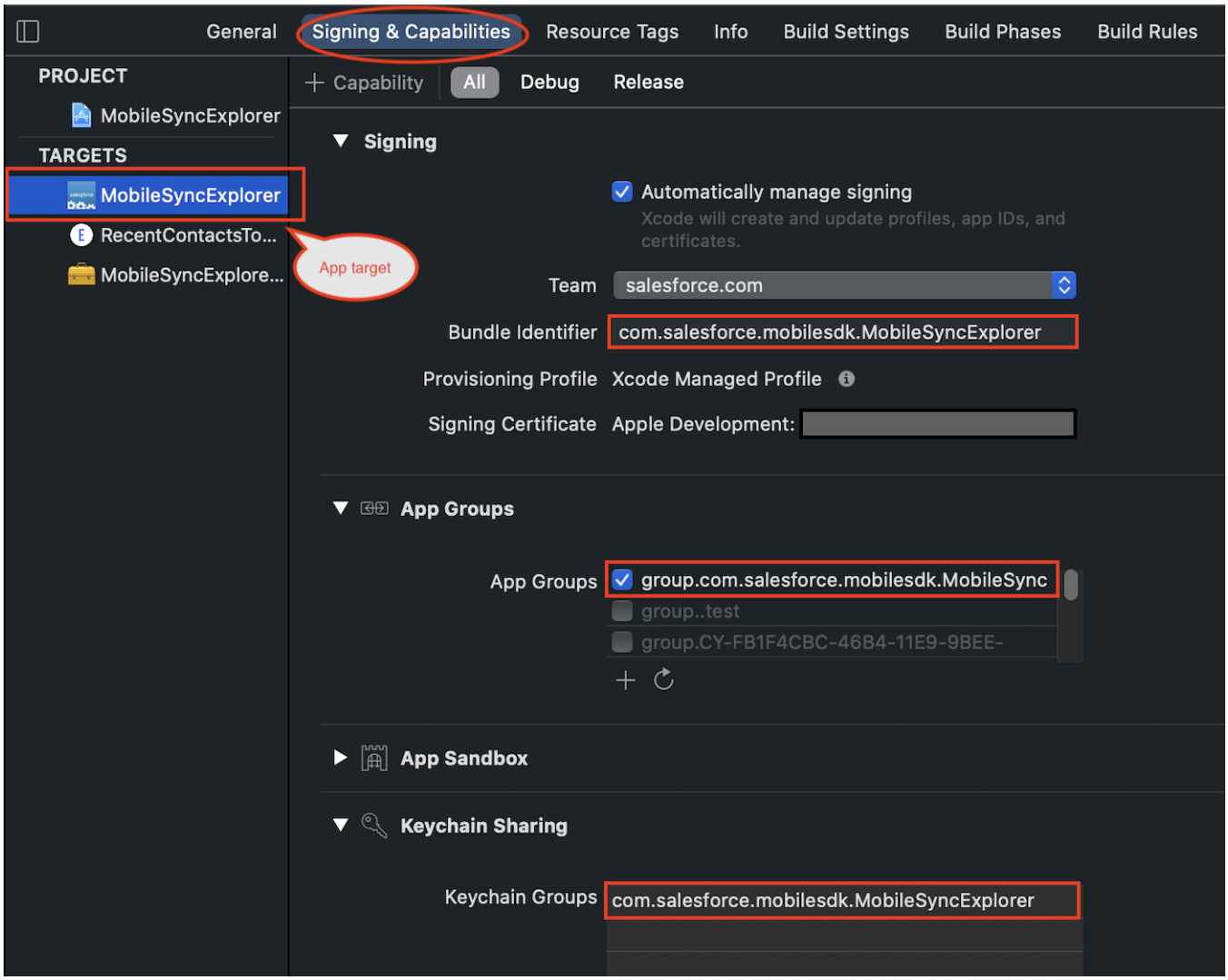
Project Settings

1. In your Mobile SDK app, create an extension target as described in the Apple developer documentation. How you handle this step and the type of extension is between you and iOS.
2. After you've created the target, select the top-level node of your Mobile SDK workspace in the Xcode Project Navigator. This step opens your project in the Project Editor.
3. In both app and extension targets, sign in to the same developer team.
4. Click **General**, and then specify a unique bundle identifier for the app target. Here's how it looks in Xcode 12.4:

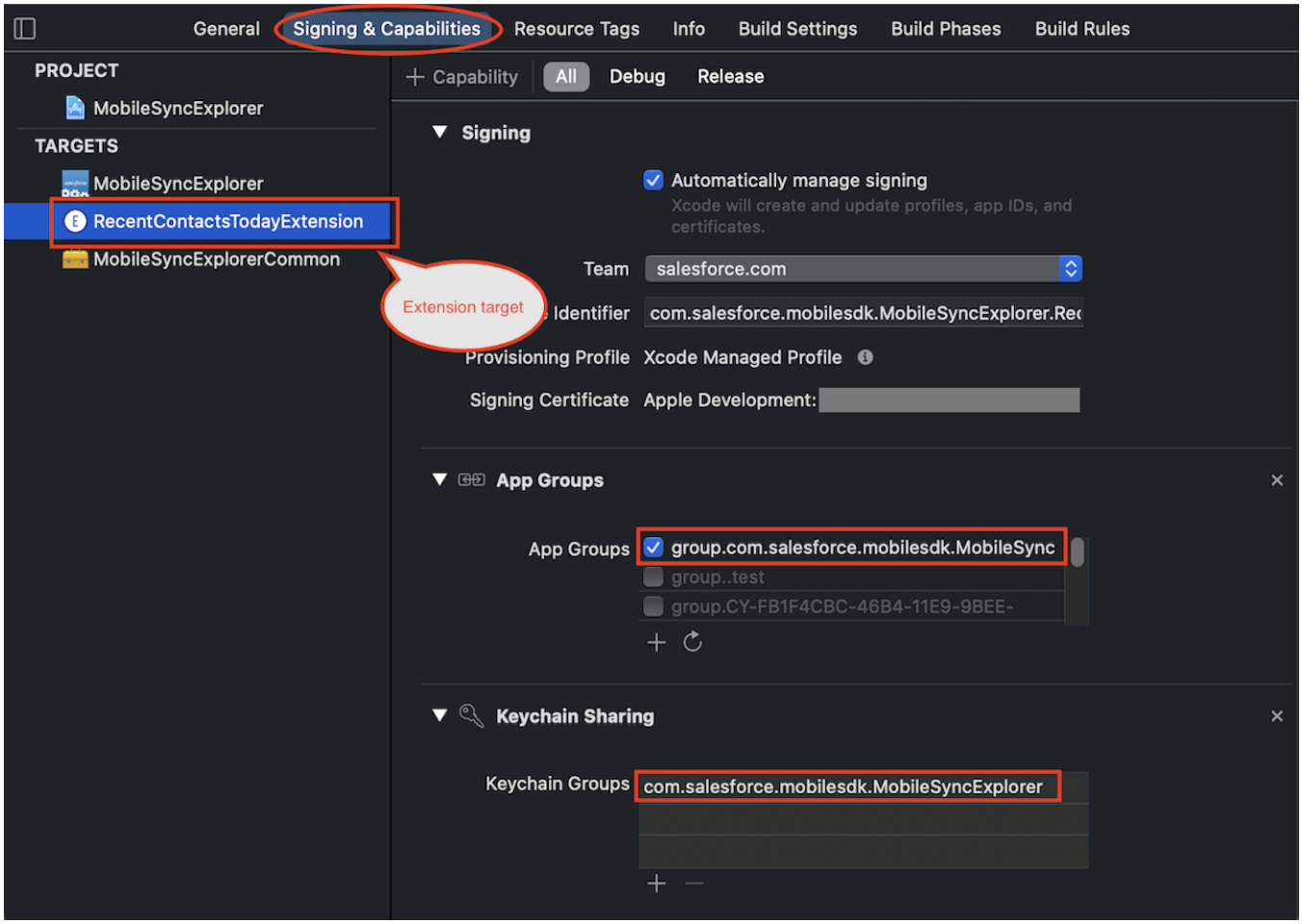


5. Repeat the bundle identifier step for the extension target. This identifier must also be unique.
6. In your app configuration, select your Mobile SDK app target and then click **Capabilities**.
7. Turn on **App Groups** and **Keychain Sharing** in your app target.
8. Under **App Groups**, select or create an app group. Use a unique label that identifies your app, such as "group.myapp.shared".

9. Under **Keychain Sharing**, select or create a keychain group. Use a unique label that identifies your app, such as "com.myapp.MyApp".



10. Repeat the App Groups and Keychain Sharing steps for your extension target. The two values in the extension target must exactly match the corresponding values in the application target.



App Group Settings

When you incorporate an iOS app extension into a forceios app, you add code that tells Mobile SDK that you're working in an app group. You add these lines in two places: the `AppDelegate` class in your main app, and the view controller or scene delegate class in your app extension.

Mobile SDK provides a utility class, `SFSDKDatasharingHelper`, that stores the necessary properties. Values you set for these properties in the main app and the app extension must be identical.

After the user attempts to log in to Salesforce, you store the authentication status where the extension code can access it. To do so, use an `NSUserDefaults` dictionary whose suite name matches your app group name. Mobile SDK provides a `NSUserDefaults (SFAdditions)` category that provides such a dictionary. This category defines one class method, `msdkUserDefaults`, that returns a group dictionary if the `appGroupEnabled` flag of the singleton `SFSDKDatasharingHelper` object is set to YES. Here's the SDK category:

```
@implementation NSUserDefaults (SFAdditions)

+ (NSUserDefaults *)msdkUserDefaults {
    NSUserDefaults *sharedDefaults = nil;
    if ([SFSDKDatasharingHelper sharedInstance].appGroupEnabled) {
```

```
        sharedDefaults = [[NSUserDefaults alloc]
            initWithSuiteName:[SFSDKDatasharingHelper sharedInstance].appGroupName];
    } else {
        sharedDefaults = [NSUserDefaults standardUserDefaults];
    }
    return sharedDefaults;
}

@end
```

AppDelegate Code Changes

The following steps apply to the `init` method of your main app's `AppDelegate` class.

1. At the top of the `init` method, set `appGroupName` and `appGroupEnabled` on the `SFSDKDatasharingHelper` shared instance.

```
...
- (id)init
{
    self = [super init];
    if (self) {
     // Insert these two lines, using your app group name
        [SFSDKDatasharingHelper sharedInstance].appGroupName =
            @"<your app group name>";
        [SFSDKDatasharingHelper sharedInstance].appGroupEnabled = YES;
...
```

2. After the user has successfully logged in, store a value reflecting the login status in an object that the extension code can access. You can infer the status by reading the `currentUser` property of the `SFUserAccountManager` object. To safely and consistently update the login status, set this property in the callback method of the `registerBlockForCurrentUserChangeNotifications` call in the `init` method. The following example uses the Mobile SDK `NSUserDefaults (SFAdditions)` category to store the Boolean value under a key named `userLoggedIn`.

```
[SFSDKAuthHelper registerBlockForCurrentUserChangeNotifications:^{
    __strong typeof (weakSelf) strongSelf = weakSelf;
    BOOL loggedIn = [SFUserAccountManager.sharedInstance currentUser] != nil;
    [[NSUserDefaults msdkUserDefaults] setBool:loggedIn forKey:@"userLoggedIn"];
    [strongSelf resetViewState:^{
        [strongSelf setupRootViewController];
    }];
}];
```

App Extension Code Changes

At runtime, your iOS app extension operates as a second app, so you have to "bootstrap" it as well. You apply the same `appGroupName` and `appGroupEnabled` changes as you did in the main app's `AppDelegate` class.

App extensions can't perform authentication tasks such as user logins. However, before making calls to a Mobile SDK manager such as `MobileSyncSDKManager`, you must verify that a user has logged in. You do this verification by checking the value stored by the main app in the `NSUserDefaults` dictionary.

1. In your app extension's initialization entry point, set and enable the app group.

```
[SFSDKDatasharingHelper sharedInstance].appGroupName = @"<your app group name>";
[SFSDKDatasharingHelper sharedInstance].appGroupEnabled = YES;
```

2. Verify that the `userLoggedIn` value indicates success. Continue with other Mobile SDK calls only if `userLoggedIn` equals YES.

If the app determines that a user has successfully logged in, the app extension can apply that user's shared credentials to Salesforce API calls.

Important:

- It's the developer's responsibility to determine the user's login status. The iOS app extension code must not attempt to invoke the `SalesforceSDKManager` object before the user successfully logs in.

Example: The following code is taken from the MobileSyncExplorer native sample app. This sample defines an app extension that displays a list of MRU Contact records.

Besides the app and extension targets, the MobileSyncExplorer project defines a third target, `MobileSyncExplorerCommon`. This container module is accessible in main and extension targets but isn't part of the app group. It implements a `MobileSyncExplorerConfig` class that declares and manages three properties:

appGroupName
The "app group" name, as defined in the project settings. This string is also hard-coded in the `MobileSyncExplorerConfig` class.

appGroupsEnabled
Boolean that indicates whether app groups are enabled.

userLogInStatusKey
The name of the key used to store the user login status in the group `NSUserDefaults` dictionary.

At runtime, both the main app and the extension use the values `appGroupName` and `appGroupsEnabled` properties to set matching properties in their own copies of the `SFSDKDataSharingHelper` class. When properly configured, this class enables both apps to use the login credentials of the current user to make calls to the Salesforce API.

To retain the user login status, MobileSyncExplorer defines a private method named `resetUserloginStatus`. This method stores the given value in the user defaults dictionary under the `userLoggedIn` key.

```
- (void)resetUserloginStatus {
    BOOL loggedIn = [SFUserAccountManager.sharedInstance currentUser] != nil;
    [[NSUserDefaults msdkUserDefaults] setBool:loggedIn forKey:@"userLoggedIn"];
    [SFSDKMobileSyncLogger log:[self class] level:SFLogLevelDebug format:@"%d
userLoggedIn", [[NSUserDefaults msdkUserDefaults] boolForKey:@"userLoggedIn"] ];
}
```

The `init` method uses a call to `resetUserloginStatus` to simplify the `registerBlockForCurrentUserChangeNotifications` block.

```
- (id)init
{
    self = [super init];
    if (self) {
        MobileSyncExplorerConfig *config = [MobileSyncExplorerConfig sharedInstance];

        [SFSDKDatasharingHelper sharedInstance].appGroupName = config.appGroupName;
        [SFSDKDatasharingHelper sharedInstance].appGroupEnabled =
```

```
config.appGroupsEnabled;

        [MobileSyncSDKManager initializeSDK];

        //App Setup for any changes to the current authenticated user
        __weak typeof (self) weakSelf = self;
        [SFSDKAuthHelper registerBlockForCurrentUserChangeNotifications:^{
            __strong typeof (weakSelf) strongSelf = weakSelf;
            [strongSelf resetUserloginStatus];
            [strongSelf resetViewState:^{
                [strongSelf setupRootViewController];
            }];
        }];
    }
    return self;
}
```

Notice that this code uses an instance of `MobileSyncExplorerConfig` to store the `appGroupName` and `appGroupsEnabled` properties.

The app extension is implemented in the `RecentContactsTodayExtension/TodayViewController.m` file. For a Today extension, the entry point to custom code is `widgetPerformUpdateWithCompletionHandler:`. The app extension reads the `userLogInStatusKey` value, which it queries through its custom `userIsLoggedIn` "getter" method. Notice that this getter method retrieves the app group information where the main app stored them—in the shared `MobileSyncExplorerConfig` object.

```
- (void)widgetPerformUpdateWithCompletionHandler:(void
(^)(NCUpdateResult))completionHandler {
    MobileSyncExplorerConfig *config = [MobileSyncExplorerConfig sharedInstance];
    [SFSDKDatasharingHelper sharedInstance].appGroupName = config.appGroupName;
   [SFSDKDatasharingHelper sharedInstance].appGroupEnabled = config.appGroupsEnabled;


    if ([self userIsLoggedIn] ) {
       [SFLogger log:[self class] level:SFLogLevelError format:@"User has logged in"];

        [MobileSyncSDKManager initializeSDK];
       SFUserAccount *currentUser = [SFUserAccountManager sharedInstance].currentUser;

        __weak typeof(self) weakSelf = self;
        void (^completionBlock)(void) = ^{
            [weakSelf refreshList];
        };
        if(currentUser) {
            if (!self.dataMgr) {
                self.dataMgr = [[SObjectDataManager alloc]
initWithDataSpec:[ContactSObjectData dataSpec]];
            }
            [self.dataMgr lastModifiedRecords:kNumberOfRecords
completion:completionBlock];
        }
    }
    completionHandler(NCUpdateResultNewData);
}
```

```
- (BOOL)userIsLoggedIn {
    MobileSyncExplorerConfig *config = [MobileSyncExplorerConfig sharedInstance];
    return [[NSUserDefaults msdkUserDefaults] boolForKey:config.userLogInStatusKey];
}
```

Profiling with Signposts

Signposts provide a powerful option for logging your app's runtime resource usage. When viewed in Xcode's Instruments, signpost logs allow you to model your app's performance over time. These profiles help you find bottlenecks and other anomalies in your code. Beginning in version 7.1, Mobile SDK add signposts in heavily used portions of its code so you can profile its performance in your app. Mobile SDK signposts are available for Swift and Objective-C apps.

In debug profiles, enabling signposts requires a single flip of an Xcode switch. For production profiles in apps that use CocoaPods, you make a few changes to the Podfile and then rebuild.

Using Signposts in Debug Builds

In Mobile SDK libraries, signposts are pre-configured for Debug builds. However, Xcode profile schemes default to Release builds. Let's change that.

To profile a debug build:

1. In the Xcode taskbar, click the scheme name and select **Edit Scheme**.
2. Select **Profile**.
3. Set Build Configuration to **Debug**.

Using Signposts in Production Builds

To profile a production build with Mobile SDK signposts, you don't edit the default profile scheme. If you've already tinkered with default settings, however, verify them as follows:

1. In the Xcode taskbar, click the scheme name and select **Edit Scheme**.
2. Select **Profile**.
3. Make sure that Build Configuration is set to **Release**.

If you created your project "manually", without using forceios or CocoaPods, your configuration is ready to run. Skip to Running a Signpost Profile Build.

If you used forceios to create your app, CocoaPods controls the settings for Mobile SDK pods. To profile a production build of a Mobile SDK app that uses CocoaPods:

1. Edit the following section of your app's Podfile.

```
# Comment the following if you do not want the SDK to emit signpost
# events for instrumentation. Signposts are  enabled for non release version of the app.

 post_install do |installer_representation|
       installer_representation.pods_project.targets.each do |target|
           target.build_configurations.each do |config|
               if config.name == 'Debug'
```

```
                    config.build_settings['GCC_PREPROCESSOR_DEFINITIONS'] ||=
['$(inherited)', 'DEBUG=1','SIGNPOST_ENABLED=1']
                    config.build_settings['OTHER_SWIFT_FLAGS'] = ['$(inherited)',
'-DDEBUG', '-DSIGNPOST_ENABLED']
                end
            end
        end
    end
```

- Change `if config.name == 'Debug'` (as shown here) to `if config.name == 'Release'`.
- Remove the `'$(inherited)'`, `'DEBUG=1'` and `'-DDEBUG'` settings.

Here's the intended result.

```
# Comment the following if you do not want the SDK to emit signpost
# events for instrumentation. Signposts are  enabled for non release version of the app.

 post_install do |installer_representation|
        installer_representation.pods_project.targets.each do |target|
            target.build_configurations.each do |config|
                if config.name == 'Release'
                    config.build_settings['GCC_PREPROCESSOR_DEFINITIONS'] ||=
['SIGNPOST_ENABLED=1']
                    config.build_settings['OTHER_SWIFT_FLAGS'] = ['-DSIGNPOST_ENABLED']
                end
            end
        end
    end
```

2. **Important:** Close your workspace.
3. In the Terminal app, switch to your workspace directory and run: *`pod update`*
4. Reopen your workspace in Xcode.

Running a Signpost Profile Build

To run a signpost build:

1. In the Xcode menu, click **Product** > **Profile**
2. When prompted, select a Profiling Template—for example, "Blank"—and click **Choose**.
3. In the Instruments panel, click **+** and find "os_signposts".
4. Double-click **os_signposts**.
5. Click **Record**. Enjoy the show!

iOS Sample Applications

A native app you create with forceios is itself a sample application, though limited in scope. The native iOS sample apps demonstrate more functionality you can examine and work into your own apps.

- **MobileSyncExplorer** demonstrates the power of the native Mobile Sync library on iOS. It resides in Mobile SDK for iOS under `native/SampleApps/MobileSyncExplorer`.

- **RestAPIExplorer** exercises all native REST API wrappers. It resides in Mobile SDK for iOS under `native/SampleApps/RestAPIExplorer`.

Mobile SDK provides iOS wrappers for the following hybrid apps.

- **AccountEditor**: Demonstrates how to synchronize offline data using the `mobilesync.js` library.
- **MobileSyncExplorer**: Demonstrates how to synchronize offline data using the Mobile Sync plugin.
- **NoteSync**: Demonstrates how to use non-REST APIs to retrieve Salesforce Notes.

CHAPTER 7 Native Android Development

In this chapter ...

Salesforce Mobile SDK for Android provides source code, build scripts, and native sample apps to get you up and running. It gives you template apps that implement two basic features of any Mobile SDK app:

- Automation of the OAuth2 authentication flow, making it easy to integrate the process with your app.
- Access to the Salesforce REST API, with utility classes that simplify that access.

Sample native applications show you basic techniques for implementing login, REST API calls, and other Mobile SDK features.

Android Native Quick Start

Use the following procedure to get started quickly.

1. Make sure you meet all of the native Android requirements.
2. Install Mobile SDK for Android. If you prefer, you can install Mobile SDK from the Mobile SDK GitHub Repositories instead.
3. At the command line, run the forcedroid application to create a new Android project, and then run that app in Android Studio or from the command line.
4. Follow the instructions at Setting Up Sample Projects in Android Studio.

Native Android Requirements

Salesforce Requirements

- Salesforce Mobile SDK 11.1 or later for Android.
- A Salesforce Developer Edition organization with a connected app.

The `SalesforceSDK` project is built with the Android Nougat (API 24) library.

Android Requirements

To install the Android development environment, go to the Mobile SDK Trailhead project and do the following steps:

1. Install Common Components
2. Set Up Your Android Development Environment

Here's a list of the items you install.

- Java JDK 11.0.11+9 or later—www.oracle.com/downloads.
- Latest version of Android Studio —developer.android.com/sdk.
- Android SDK, including all APIs in the following range:
  - Minimum API: Android Nougat (API 24)
  - Target API: Android 13 (API 33)
- Android SDK Tools
- Android Virtual Device (AVD)

Tip:

- For best results, install all Android SDK versions recommended by the Android SDK Manager, and all available versions of Android SDK tools.
- On Windows, be sure to run Android Studio as administrator.

See Also

"android studio" at developer.android.com

Creating an Android Project with Forcedroid

To create an app, use forcedroid in a terminal window or at a Windows command prompt. The forcedroid utility gives you two ways to create your app.

- Specify the type of application you want, along with basic configuration data.

  OR

- Use an existing Mobile SDK app as a template. You still provide the basic configuration data.

You can use forcedroid in interactive mode with command-line prompts, or in script mode with command-line arguments. To see command usage information, type *`forcedroid`* without arguments.

Forcedroid Project Types

The *`forcedroid create`* command requires you to specify one of the following project types:

App Type	Architecture	Language
`native`	Native	Java
`native_kotlin` (or just press *`RETURN`*)	Native	Kotlin

To develop a native Android app in Java, specify *`native`*.

Using `forcedroid create` Interactively

To enter application options interactively at a command prompt, type *`forcedroid create`*. The forcedroid utility then prompts you for each configuration option. For example:

```
$ forcedroid create
Enter your application type (native, native_kotlin): <press RETURN>
Enter your application name: testNative
Enter the package name for your app (com.mycompany.myapp): com.bestapps.android
Enter your organization name (Acme, Inc.): BestApps
Enter output directory for your app (leave empty for the current directory): testNative
```

This command creates a native Kotlin Android app named "testNative" in the `testNative\` subdirectory of your current directory.

Using `forcedroid create` in Script Mode

In script mode, you can use forcedroid without interactive prompts. For example, to create a native app written in Java:

```
$ forcedroid create --apptype="native" --appname="package-test"
--packagename="com.acme.mobile_apps"
    --organization="Acme Widgets, Inc." --outputdir="PackageTest"
```

Or, to create a native app written in Kotlin:

```
$ forcedroid create --apptype="native_kotlin" --appname="package-test"
--packagename="com.acme.mobile_apps"
    --organization="Acme Widgets, Inc." --outputdir="PackageTest"
```

Each of these calls creates a native app named “package-test” and places it in the `PackageTest/` subdirectory of your current directory.

Creating an App from a Template

The *`forcedroid createWithTemplate`* command is identical to *`forcedroid create`* except that it asks for a GitHub repo URI instead of an app type. You set this URI to point to any repo directory that contains a Mobile SDK app that can be used as a template. Your template app can be any supported Mobile SDK app type. The script changes the template’s identifiers and configuration to match the values you provide for the other parameters.

Before you use `createWithTemplate`, it's helpful to know which templates are available. To find out, type *`forcedroid listtemplates`*. This command prints a list of templates provided by Mobile SDK. Each listing includes a brief description of the template and its GitHub URI. For example:

```
Available templates:

1) Basic Kotlin application
forcedroid createwithtemplate --templaterepouri=AndroidNativeKotlinTemplate
2) Basic Java application
forcedroid createwithtemplate --templaterepouri=AndroidNativeTemplate
3) Sample Kotlin Identity Provider application
forcedroid createwithtemplate --templaterepouri=AndroidIDPTemplate
```

Once you've found a template's URI, you can plug it into the forcedroid command line. Here’s command-line usage information for `forcedroid createWithTemplate`:

```
Usage:
forcedroid createWithTemplate
    --templaterepouri=<Template repo URI> (e.g.
https://github.com/forcedotcom/SalesforceMobileSDK-Templates/MobileSyncExplorerReactNative)]

    --appname=<Application Name>
    --packagename=<App Package Identifier> (e.g. com.mycompany.myapp)
    --organization=<Organization Name> (Your company's/organization's name)
    --outputdir=<Output directory> (Leave empty for current directory)]
```

For any template in the `SalesforceMobileSDK-Templates` repo, you can drop the path for `templaterepouri`—just the template name will do. For example:

```
forcedroid createwithtemplate --templaterepouri=AndroidNativeKotlinTemplate
```

You can use `forcedroid createWithTemplate` interactively or in script mode. For example, here’s a script mode call:

```
forcedroid createWithTemplate
--templaterepouri=SalesforceMobileSDK-Templates/AndroidIDPTemplate#v6.2.0
--appname=MyIDP-Android
--packagename=com.mycompany.react
--organization="Acme Software, Inc."
--outputdir=testWithTemplate
```

This call creates an Android identity provider app with the same source code and resources as the Android IdP sample app. Forcedroid places the new app in the `testWithTemplate/` subdirectory of your current directory. It also changes the app name to "MyIDP-Android" throughout the project.

Checking the Forcedroid Version

To find out which version of forcedroid you've installed, run the following command:

```
forcedroid version
```

Import and Build Your App in Android Studio

1. Open the project in Android Studio.
   - From the Welcome screen, click **Import Project (Eclipse ADT, Gradle, etc.)**.

     OR

   - From the File menu, click **File** > **New** > **Import Project...**.
2. Browse to your project directory and click **OK**.

   Android Studio automatically builds your workspace. This process can take several minutes. When the status bar reports "Gradle build successful", you're ready to run the project.
3. Click **Run <*project_name*>**, or press SHIFT+F10. For native projects, the project name is the app name that you specified.

   Android Studio launches your app in the emulator or on your connected Android device.

Building and Running Your App from the Command Line

After the command-line returns to the command prompt, the forcedroid script prints instructions for running Android utilities to configure and clean your project. Follow these instructions if you want to build and run your app from the command line.

1. Build the new application.
   - **Windows:**

     ```
     cd <your_project_directory>
     gradlew assembleDebug
     ```
   - **Mac:**

     ```
     cd <your_project_directory>
     ./gradlew assembleDebug
     ```

   When the build completes successfully, you can find your signed APK debug file in the project's `build/outputs/apk` directory.
2. If you're using an emulator that isn't running, use the Android AVD Manager to start it. If you're using a physical device, connect it.
3. Install the APK file on the emulator or device.
   - **Windows:**

     ```
     adb install <path_to_your_app>\build\outputs\apk\<app_name>.apk
     ```

- **Mac:**

```
./adb install <path_to_your_app>/build/outputs/apk/<app_name>.apk
```

If you can't find your newly installed app, try restarting your emulator or device. For more information, search "Build your app from the command line" at developer.android.com.

How the Forcedroid Script Generates New Apps

- The script downloads templates at runtime from a GitHub repo.
- The `forcedroid create` command uses the default Kotlin template in the SalesforceMobileSDK-Templates GitHub repo.
- Generated apps use Gradle.
- The script uses npm at runtime to download Mobile SDK libraries. The `settings.gradle` file points to these libraries under `node_modules`.

SEE ALSO:

Updating Mobile SDK Apps (5.0 and Later)

Using a Custom Template to Create Apps

About Mobile SDK Templates

Mobile SDK defines a template for each architecture it supports on iOS and Android. These templates are maintained in the github.com/forcedotcom/SalesforceMobileSDK-Templates repo. When a customer runs the forcedroid or forceios *`create`* command, the script copies the appropriate built-in template from the repo and transforms this copy into the new app. Apps created this way are basic Mobile SDK apps with little functionality.

Perhaps you'd like to create your own template, with additional functionality, resources, or branding. You can harness the same Mobile SDK mechanism to turn your own app into a template. You can then tell forcedroid or forceios to use that template instead of its own.

How to Use a Custom Template

In addition to forcedroid and forceios *`create`*, Mobile SDK defines a *`createWithTemplate`* command.When you run forcedroid or forceios *`createWithTemplate`*, you specify a template app repo instead of an app type, followed by the remaining app creation parameters. The template app repo contains a Mobile SDK app that the script recognizes as a template. To create a new Mobile SDK app from this template, the script copies the template app to a new folder and applies your parameter values to the copied code.

The `template.js` File

To accept your unknown app as a template, forceios and forcedroid require you to define a `template.js` configuration file. You save this file in the root of your template app repo. This file tells the script how to perform its standard app refactoring tasks—moving files, replacing text, removing and renaming resources. However, you might have even more extensive changes that you want to apply. In such cases, you can also adapt `template.js` to perform customizations beyond the standard scope. For example, if you insert your app name in classes other than the main entry point class, you can use `template.js` to perform those changes.

A `template.js` file contains two parts: a JavaScript "prepare" function for preparing new apps from the template, and a declaration of exports.

The `template.js` Prepare Funtion

Most of a `template.js` file consists of the "prepare" function. By default, prepare functions use the following signature:

```
function prepare(config, replaceInFiles, moveFile, removeFile)
```

You can rename this function, as long as you remember to specify the updated name in the list of exports. The Mobile SDK script calls the function you export with the following arguments:

- `config`: A dictionary identifying the platform (iOS or Android), app name, package name, organization, and Mobile SDK version.
- `replaceInFiles`: Helper function to replace a string in files.
- `moveFile`: Helper function to move files and directories.
- `removeFile`: Helper function to remove files and directories.

The default prepare function found in Mobile SDK templates replaces strings and moves and removes the files necessary to personalize a standard template app. If you intend to add functionality, place your code within the prepare function. Note, however, that the helper functions passed to your prepare function can only perform the tasks of a standard template app. For custom tasks, you'll have to implement and call your own methods.

Exports Defined in `template.js`

Each `template.js` file defines the following two exports.

appType
Assign one of the following values:

- `'native'`
- `'native_kotlin'` (forcedroid only)
- `'native_swift'` (forceios only)
- `'react_native'`
- `'hybrid_local'`
- `'hybrid_remote'`

prepare
The handle of your prepare function (listed without quotation marks).

Here's an example of the export section of a `template.js` file. This template is for a native app that defines a prepare function named `prepare`:

```
//
// Exports
//
module.exports = {
    appType: 'native',
    prepare: prepare
};
```

In this case, the prepare function's handle is, in fact, "prepare":

```
function prepare(config, replaceInFiles, moveFile, removeFile)
```

Template App Identification in `template.js` (Native and React Native Apps)

For native and React native apps, a template app's prepare function defines an app name, a package name, and an organization or company name. These values identify the template app itself—not a new custom app created from the template. At runtime, the Mobile SDK script uses these values to find the strings to be replaced with the script's input values. Here's an example of the settings for these `iOSNativeTemplate` template app:

```
// Values in template
var templateAppName = 'iOSNativeTemplate';
var templatePackageName = 'com.salesforce.iosnativetemplate';
var templateOrganization = 'iOSNativeTemplateOrganizationName';
```

Examples of template.js Files

Mobile SDK defines its own templates in the github.com/forcedotcom/SalesforceMobileSDK-Templates repo. Each template directory includes a `template.js` file. Templates include:

- `iOSNativeTemplate` (forceios only)
- `iOSNativeSwiftTemplate` (forceios only)
- `ReactNativeTemplate`
- `HybridLocalTemplate`
- `HybridRemoteTemplate`
- `AndroidNativeTemplate` (forcedroid only)
- `AndroidNativeKotlinTemplate` (forcedroid only)

These templates are "bare bones" projects used by the Mobile SDK npm scripts to create apps; hence, their level of complexity is intentionally low. If you're looking for more advanced templates, see

- `MobileSyncExplorerReactNative`
- `MobileSyncExplorerSwift`
- `AndroidIDPTemplate`
- `iOSIDPTemplate`

You can get a list of these templates with their repo paths from the `listtemplates` command. All Mobile SDK npm scripts—forcedroid, forceios, forcehybrid, and forcereact—support this command.

Note: Always match the script command to the template. Use iOS-specific templates with *`forceios createWithTemplate`* only, and Android-specific templates with *`forcedroid createWithTemplate`* only. This restriction doesn't apply to hybrid and React native templates.

Define a Basic `template.js` File

The following steps describe the quickest way to create a basic `template.js` file.

1. Copy a `template.js` file from the github.com/forcedotcom/SalesforceMobileSDK-Templates repo to the root of your custom template app repo. Be sure to choose the template that matches the type of app your template should build.
2. For native or React native apps only, update the app name, package name, and organization to reflect your template app.
3. If necessary, update the `appType` and `prepare` settings in the `module.exports` object, as described earlier. Although this step isn't required for this basic example, you might need it later if you create your own `template.js` files.

Restrictions and Guidelines

A few restrictions apply to custom templates.

- The template app can be any valid Mobile SDK app that targets any supported platform and architecture.
- A primary requirement is that the template repo and your local Mobile SDK repo must be on the same Mobile SDK version. You can use git version tags to sync both repos to a specific earlier version, but doing so isn't recommended.
- Always match the script command to the template. Use iOS-specific templates with `forceios createWithTemplate` only, and Android-specific templates with `forcedroid createWithTemplate` only. This restriction doesn't apply to hybrid and React native templates.

Setting Up Sample Projects in Android Studio

The SalesforceMobileSDK-Android GitHub repository contains sample apps you can build and run.

1. If you haven't already done so, clone the SalesforceMobileSDK-Android GitHub repository.
   - **Mac:**

     ```
     git clone https://github.com/forcedotcom/SalesforceMobileSDK-Android.git
     ./install.sh
     ```

   - **Windows:**

     ```
     git clone https://github.com/forcedotcom/SalesforceMobileSDK-Android.git
     cscript install.vbs
     ```

2. Open the project in Android Studio.
   - From the Welcome screen, click **Import Project (Eclipse ADT, Gradle, etc.)**.

     OR

   - From the File menu, click **File** > **New** > **Import Project...**.

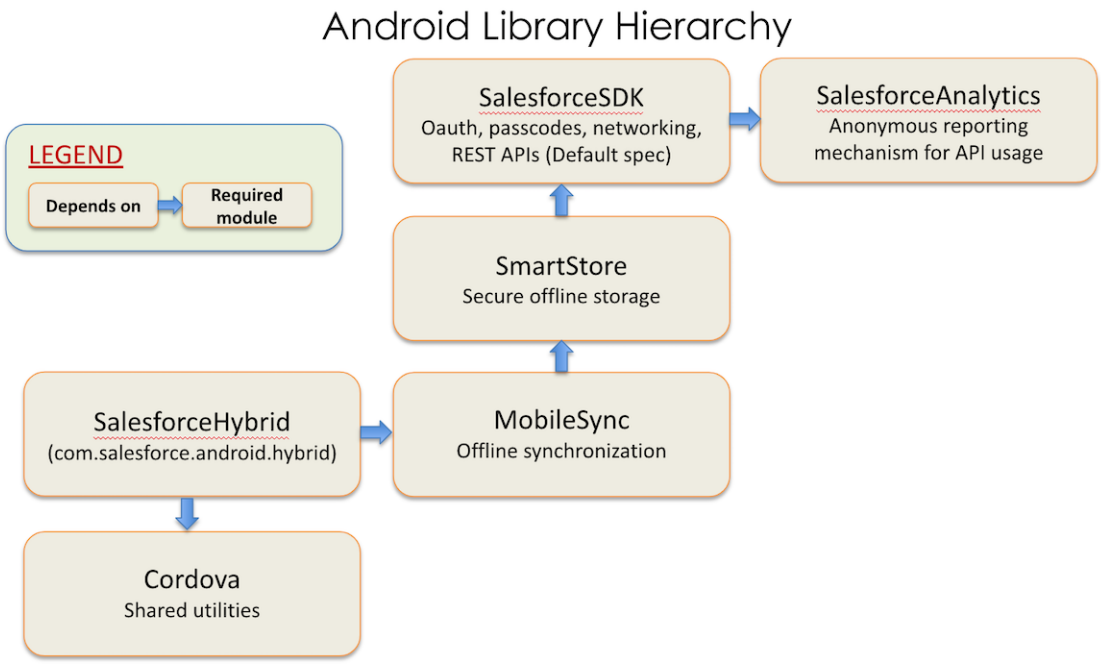
3. Browse to `<path_to_SalesforceMobileSDK-Android>/native/NativeSampleApps/` or `<path_to_SalesforceMobileSDK-Android>/hybrid/HybridSampleApps/`
4. Select one of the listed sample apps and click **OK**..
5. When the project finishes building, select the sample project in the Select Run/Debug Configurations drop-down menu.
6. Press `SHIFT-F10`.

Android Project Files

When you browse a native app in the Project window of Android Studio, you can find these library projects:

- `libs/SalesforceAnalytics`—Analytics project. Reports non-sensitive data on Mobile SDK app usage to Salesforce.
- `libs/SalesforceSDK`—Salesforce Mobile SDK project. Provides support for OAuth2 and REST API calls
- `libs/SmartStore`—SmartStore project. Provides an offline storage solution
- `libs/MobileSync`—Mobile Sync project. Implements offline data synchronization tools

Mobile SDK libraries reference each other in a dependency hierarchy, as shown in the following diagram.



Developing a Native Android App

The native Android version of the Salesforce Mobile SDK empowers you to create rich mobile apps that directly use the Android operating system on the host device. To create these apps, you need to understand Android development well enough to write code in Java or Kotlin that uses Mobile SDK native classes.

Android Application Structure

Native Android apps that use Mobile SDK typically require:

- An application entry point class that extends `android.app.Application`.
- At least one activity that extends `android.app.Activity`.

With Mobile SDK, the Android template apps:

- Create a stub application class that extends `android.app.Application`. In this class, the `onCreate()` method
  - Calls `SalesforceSDKManager.initNative()`.
  - (Optional) Enables IDP login services.
  - (Optional) Enables push notifications.
- Create an activity class that extends `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`. These Salesforce classes are optional but recommended choices for extending `android.app.Activity`.

Authentication and App Lifecycle Classes

The top-level `SalesforceSDKManager` class sets the stage for login, cleans up after logout, and provides a special event watcher that informs your app when a system-level account is deleted. OAuth protocols are handled automatically with internal classes.

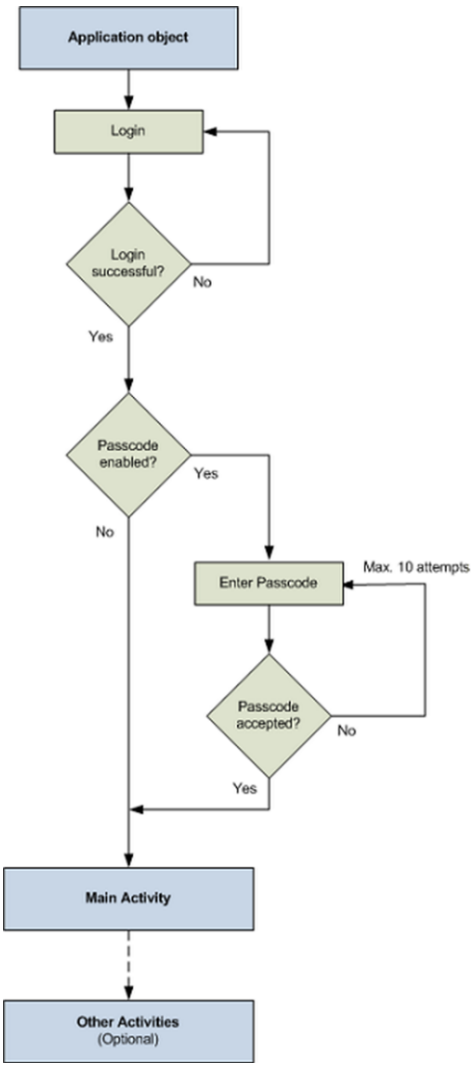
The `SalesforceActivity`, `SalesforceListActivity`, and `SalesforceExpandableListActivity` classes offer Mobile SDK implementations of standard Android UI protocols.

We recommend that you extend one of these classes for all activities in your app—not just the main activity. If you use a different base class for an activity, you're responsible for replicating the `onResume()` protocol found in `SalesforceActivity`.

Using Resources in Activities

Within your activities, you interact with Salesforce objects by calling Salesforce REST APIs. The Mobile SDK provides the `com.salesforce.androidsdk.rest` package to simplify the REST request and response flow.

You define and customize user interface layouts, image sizes, strings, and other resources in XML files. Internally, the SDK uses an `R` class instance to retrieve and manipulate your resources. However, the Mobile SDK makes its resources directly accessible to client apps, so you don't need to write code to manage these features.



Native API Packages

Salesforce Mobile SDK groups native Android APIs into Java packages. For a quick overview of these packages and points of interest within them, see Android Packages and Classes.

Overview of Native Classes

The following overviews of Mobile SDK native classes describe pertinent details of each class and give you a sense of where to find what you need.

SalesforceSDKManager Class

The `SalesforceSDKManager` class is the entry point for all native Android applications that use Salesforce Mobile SDK. It provides mechanisms for:

- Login and logout
- Passcodes
- Encryption and decryption of user data
- String conversions
- User agent access
- Application termination
- Application cleanup

Instead of calling `SalesforceSDKManager` directly, the forcedroid native template uses a subclass, `MobileSyncSDKManager`, to initialize apps.

initNative() Method

During startup, you initialize the `MobileSyncSDKManager` class object by calling its static `initNative()` method. This method takes the following arguments:

Parameter Name	Description
`applicationContext`	An instance of `Context` that describes your application's context. In an `Application` extension class, you can satisfy this parameter by passing a call to `getApplicationContext()`.
`mainActivity`	The descriptor of the class that displays your main activity. The main activity is the first activity that displays after login.

Here's an example from the `MainApplication` class of the forcedroid Java template app:

```
import com.salesforce.androidsdk.mobilesync.app.MobileSyncSDKManager;
...

public class MainApplication extends Application {
```

```
    @Override
    public void onCreate() {
        super.onCreate();
        MobileSyncSDKManager.initNative(getApplicationContext(),
            MainActivity.class);
        ...
    }
}
```

In this example, `NativeKeyImpl` is the app's implementation of `KeyInterface`. `MainActivity` subclasses `SalesforceActivity` and is designated here as the first activity to be called after login.

The Kotlin template app additionally defines a Kotlin-related constant and registers Kotlin as a used feature.

```
class MainApplication : Application() {
    companion object {
        private const val FEATURE_APP_USES_KOTLIN = "KT"
    }

    override fun onCreate() {
        super.onCreate()
        MobileSyncSDKManager.initNative(applicationContext, MainActivity::class.java)
       MobileSyncSDKManager.getInstance().registerUsedAppFeature(FEATURE_APP_USES_KOTLIN)

        ...
    }
}
```

logout() Method

The `SalesforceSDKManager.logout()` method clears user data. For example, if you've introduced your own resources that are user-specific, you can override `logout()` to keep those resources from being carried into the next user session. SmartStore, the Mobile SDK offline database, destroys user data and account information automatically at logout.

Always call the superclass `logout` method somewhere in your method override, preferably after doing your own cleanup. Here's an example of how to override `logout()`.

Kotlin

```
override fun logout(frontActivity: Activity) {
    // Clean up all persistent and non-persistent app artifacts
    // ...
    // Call superclass after doing your own cleanup
    super.logout(frontActivity)
}
```

Java

```
@Override
public void logout(Activity frontActivity) {
   // Clean up all persistent and non-persistent app artifacts
   // ...
   // Call superclass after doing your own cleanup
   super.logout(frontActivity);
}
```

getLoginActivityClass() Method

This method returns the descriptor for the login activity. The login activity defines the `WebView` through which the Salesforce server delivers the login dialog.

getUserAgent() Methods

Mobile SDK builds a user agent string to publish the app's versioning information at runtime. For example, the user agent in Mobile SDK 7.1 takes the following form.

```
SalesforceMobileSDK/<salesforceSDK version> android mobile/<android OS version>
(<device model>) <appName>/<appVersion> <appType with qualifier>
<device ID> <list of features>
```

The list of features consists of one or more two-letter descriptors of Mobile SDK features. Here's a typical example.

```
SalesforceMobileSDK/7.1.0 android mobile/9.0 (Pixel 3) RestExplorer/5.0 HybridRemote
uid_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx ftr_MU.AI.BW
```

To retrieve the user agent at runtime, call the `SalesforceSDKManager.getUserAgent()` method.

isHybrid() Method

Imagine that your Mobile SDK app creates libraries that are designed to serve both native and hybrid clients. Internally, the library code switches on the type of app that calls it, but you need some way to determine the app type at runtime. To determine the type of the calling app in code, call the boolean `SalesforceSDKManager.isHybrid()` method. True means hybrid, and false means native.

PasscodeManager Class

Mobile SDK 9.2 deprecates the `PasscodeManager` class for removal in 10.0. Mobile SDK 9.2 also reimplements the passcode flow with internal classes, so this class no longer has any effect.

Encryptor class

The `Encryptor` helper class provides static helper methods for encrypting and decrypting strings using the hashes required by the SDK. It's important for native apps to remember that all keys used by the Mobile SDK must be Base64-encoded. No other encryption patterns are accepted. Use the `Encryptor` class when creating hashes to ensure that you use the correct encoding.

Most `Encryptor` methods are for internal use, but apps are free to use this utility as needed. For example, if an app implements its own database, it can use `Encryptor` as a free encryption and decryption tool.

SalesforceActivity, SalesforceListActivity, and SalesforceExpandableListActivity Classes

`SalesforceActivity`, `SalesforceListActivity`, and `SalesforceExpandableListActivity` are the skeletal base classes for native SDK activities. They extend `android.app.Activity`, `android.app.ListActivity`, and `android.app.ExpandableListActivity`, respectively.

Each of these activity classes contains a single abstract method:

```
public abstract void onResume(RestClient client);
```

This method overloads the `Activity.onResume()` method, which is also implemented by the class. The Mobile SDK superclass delegate, `SalesforceActivityDelegate`, calls your overload when it has created a `RestClient` instance. Use this method

to cache the client that's passed in, and then use that client to perform your REST requests. For example, in the Kotlin Mobile SDK template app, the `MainActivity` class uses the following code:

```
override fun onResume(client: RestClient) {
    // Keeping reference to rest client
    this.client = client

    // Show everything
    findViewById<ViewGroup>(R.id.root).visibility = View.VISIBLE
}
```

ClientManager Class

`ClientManager` works with the Android `AccountManager` class to manage user accounts. More importantly for apps, it provides access to `RestClient` instances through two methods:

- `getRestClient()`
- `peekRestClient()`

The `getRestClient()` method asynchronously creates a `RestClient` instance for querying Salesforce data. Asynchronous in this case means that this method is intended for use on UI threads. The `peekRestClient()` method creates a `RestClient` instance synchronously, for use in non-UI contexts.

Once you get the `RestClient` instance, you can use it to send REST API calls to Salesforce.

RestClient Class

As its name implies, the `RestClient` class is an Android app's liaison to the Salesforce REST API.

You don't explicitly create new instances of the `RestClient` class. Instead, you use the `ClientManager` factory class to obtain a `RestClient` instance. Once you get the `RestClient` instance, you can use it to send REST API calls to Salesforce. The method you call depends on whether you're calling from a UI context. See ClientManager Class.

Use the following `RestClient` methods to send REST requests:

- `sendAsync()`—Call this method if you obtained your `RestClient` instance by calling `ClientManager.getRestClient()`.
- `sendSync()`—Call this method if you obtained your `RestClient` instance by calling `ClientManager.peekRestClient()`.

sendSync() Method

You can choose from three overloads of `RestClient.sendSync()`, depending on the degree of information you can provide for the request.

sendAsync() Method

The `RestClient.sendAsync()` method wraps your `RestRequest` object in a new instance of the `OkHttpClient Call` class. It then adds the `Call` object to the request queue and returns that object.

Tocancel a request while it's pending, call `cancel()` on the `Call` object. To access the underlying request queue object, use the `OkHttpClient` class. See examples at Managing the Request Queue.

RestRequest Class

The `RestRequest` class natively handles the standard Salesforce data operations offered by the Salesforce REST API and SOAP API. It creates and formats REST API requests from the data your app provides. It is implemented by Mobile SDK and serves as a factory for specialized instances of itself.

Supported operations are:

Operation or Resource	Parameters	Description
User Info		
SOQL Query	Query string, API version, batch size	Returns the requested fields of records that satisfy the SOQL query. By default, returns up to 2,000 records at once. If you specify a batch size, returns records in batches up to that size. Specifying a batch size does not guarantee that the returned batch is the requested size.
Versions	None	Returns Salesforce version metadata
Object Layout		
Batch request	API version, flag telling the batch process whether to halt in the case of error, list of subrequests	Returns a `RestRequest` object containing a batch of up to 25 subrequests specified in a list of `RestRequest` objects. Each subrequest counts against rate limits.
Resources	API version	Returns available resources for the specified API version, including resource name and URI
Metadata	API version, object type	Returns the object's complete metadata collection
DescribeGlobal	API version	Returns a list of all available objects in your org and their metadata
Describe	API version, object type	Returns a description of a single object type
Create	API version, object type, map of field names to value objects	Creates a new record in the specified object
CompositeRequest	API version, "all or none" flag that indicates whether to treat all requests as a transactional block in error conditions, hash map of subrequests (values) and their reference ID (keys)	Returns a `RestRequest` object that you use to execute the composite request. Regardless of the number of subrequests, each composite request counts as one API call. See "Composite" in the REST API Developer Guide.
Retrieve	API version, object type, object ID, list of fields	Retrieves a record by object ID

Operation or Resource	Parameters	Description
Search	API version, SOQL query string	Executes the specified SOQL search
SearchResultLayout	API version, list of objects	Returns search result layout information for the specified objects
SearchScopeAndOrder	API version	Returns an ordered list of objects in the default global search scope of a logged-in user
SObject Tree	API version, object type, list of SObject tree nodes	Returns a `RestRequest` object for an SObject tree based on the given list of SObject tree nodes.
Update	API version, object type, object ID, map of field names to value objects, `If-Unmodified-Since` date (optional)	Updates an object with the given map. For conditional updates, Mobile SDK supports `If-Unmodified-Since` requests.
Upsert	API version, object type, external ID field, external ID, map of field names to value objects	Updates or inserts an object from external data, based on whether the external ID currently exists in the external ID field. If you set the name of the external ID field to"Id" and the external ID to null, a new record is created.
Delete	API version, object type, object ID	Deletes the object of the given type with the given ID
Notification	API version, notification ID	Retrieves the notification with the given ID.
NotificationUpdate	API version, notification ID, read status, seen status	Updates the "read" (if non-null) and "seen" (if non-null) statuses of the notification with the given ID.
Notifications	API version, number of notifications, date before, date after	Retrieves notifications sent before or after a given date. "Date before" and "date after" are mutually exclusive parameters.
NotificationsStatus	API version	Retrieves the status of the current user's notifications.
NotificationsUpdate	API version, notification IDs, date before, read status, seen status	Updates the "read" (if non-null) and "seen" (if non-null) statuses of notifications with the given IDs, or those sent before the given date. IDs and "date before" are mutually exclusive parameters.

To create instances of `RestRequest`, use one of the following `RestRequest` static factory methods.

- `getRequestForUserInfo()`
- `getRequestForVersions()`

- `getRequestForResources()`
- `getRequestForDescribeGlobal()`
- `getRequestForMetadata()`
- `getRequestForDescribe()`
- `getRequestForCreate()`
- `getRequestForRetrieve()`
- `getRequestForUpdate()`
- `getRequestForUpsert()`
- `getRequestForDelete()`
- `getRequestForQuery()`
- `getRequestForSearch()`
- `getRequestForSearchScopeAndOrder()`
- `getRequestForSearchResultLayout()`
- `getRequestForObjectLayout()`
- `getBatchRequest()`
- `getCompositeRequest()`
- `getRequestForSObjectTree()`
- `getRequestForNotification()`
- `getRequestForNotificationUpdate()`
- `getRequestForNotifications()`
- `getRequestForNotificationsStatus()`
- `getRequestForNotificationsUpdate()`

Each of these methods returns a `RestRequest` object. You send your request by passing the returned `RestRequest` object to `RestClient.sendAsync()` or `RestClient.sendSync()`. See Using REST APIs.

Example: For sample calls, see `/libs/test/SalesforceSDKTest/src/com/salesforce/androidsdk/rest/RestRequestTest.java` at github.com/forcedotcom/SalesforceMobileSDK-Android.

SEE ALSO:

Supported Salesforce APIs

FileRequests Class

The `FileRequests` class provides methods that create file operation requests. Each method returns a new `RestRequest` object. Applications send this object to the Salesforce service to process the request. For example, the following code snippet calls the `ownedFilesList()` method to retrieve a `RestRequest` object. It then sends the `RestRequest` object to the server using `RestClient.sendAsync()`:

Kotlin

```
val ownedFilesRequest = FileRequests.ownedFilesList(null, null)
val client = this.client
client?.sendAsync(ownedFilesRequest, object : AsyncRequestCallback {
```

```
    // Do something with the response
})
```

Java

```
RestRequest ownedFilesRequest = FileRequests.ownedFilesList(null, null);
RestClient client = this.client;
client.sendAsync(ownedFilesRequest, new AsyncRequestCallback() {
    // Do something with the response
});
```

Note: This example passes null to the first parameter (`userId`). This value tells the `ownedFilesList()` method to use the ID of the context, or logged in, user. The second null, for the `pageNum` parameter, tells the method to fetch the first page of results.

See Files and Networking for a full description of `FileRequests` methods.

Methods

For a full reference of `FileRequests` methods, see Package com.salesforce.androidsdk.rest.files. For a full description of the REST request and response bodies, go to **Connect REST API Developer Guide** > **Resources** > **Files Resources** at http://www.salesforce.com/us/developer/docs/chatterapi.

OkHttp: The Underlying Network Library

Beginning with Mobile SDK 4.2, the Android REST request system uses OkHttp (v3.2.0), an open-source external library from Square Open Source, as its underlying architecture. This library replaces the Google Volley library from past releases. As a result, Mobile SDK no longer defines the `WrappedRestRequest` class.

Example: The following examples show how to perform some common network operations with `OkHttpClient`.

Common Imports

```
import okhttp3.Headers;
import okhttp3.HttpUrl;
import okhttp3.OkHttpClient;
import okhttp3.Call;
import okhttp3.Dispatcher;
import okhttp3.Request;
import okhttp3.RequestBody;
import okhttp3.Response;
```

Obtain the Current OkHttp Client Handle

To get the handle of the `OkHttpClient` that the current `RestClient` instance is using:

Kotlin

```
var okClient = restClient.getOkHttpClient()
```

Java

```
OkHttpClient okClient = restClient.getOkHttpClient();
```

Obtain the OkHttp Dispatcher

Kotlin

```
var dispatcher = restClient.getOkHttpClient().dispatcher()
```

Java

```
Dispatcher dispatcher = restClient.getOkHttpClient().dispatcher();
```

Cancel All Pending Calls

Kotlin

```
var dispatcher = restClient.getOkHttpClient().dispatcher()
dispatcher.cancelAll()
```

Java

```
Dispatcher dispatcher = restClient.getOkHttpClient().dispatcher();
                                dispatcher.cancelAll();
```

Store the OkHttp Handle to a REST Request

Kotlin

```
var call = restClient.sendAsync(restRequest, callback)
```

Java

```
Call call = restClient.sendAsync(restRequest, callback);
```

Cancel a Specific REST Request Using a Stored Handle

Kotlin

```
var call = restClient.sendAsync(restRequest, callback)
```

Java

```
Call call = restClient.sendAsync(restRequest, callback);
                                ...
                                call.cancel();
```

For more information, see square.github.io/okhttp/.

UI Classes

Activities in the `com.salesforce.androidsdk.ui` package represent the UI resources that are common to all Mobile SDK apps. You can style, skin, theme, or otherwise customize these resources through XML. If you override these resources, you're responsible for maintaining them when Mobile SDK breaks compatibility. With the exceptions of `SalesforceActivity`, `SalesforceListActivity`, and `SalesforceExpandableListActivity`, do not override these activity classes with intentions of replacing the resources at runtime.

LoginActivity Class

`LoginActivity` defines the login screen. The login workflow is worth describing because it explains two other classes in the activity package. In the login activity, if you press the Menu button, you get three options: **Clear Cookies**, **Reload**, and **Change Server**. **Change Server** launches an instance of the `ServerPickerActivity` class, which displays **Production** and **Sandbox** servers and an **Add Connection** option. When a user clicks**Add Connection**, `ServerPickerActivity` launches an instance of the `CustomServerURLEditor` class. This class displays a popover dialog that lets you type in the name of the custom server.

Other UI Classes

`SalesforceDroidGapActivity` and `SalesforceGapViewClient` are used only in hybrid apps. These classes don't affect your native API development efforts.

Utility Classes

Though most of the classes in the `util` package are for internal use, several of them can also benefit third-party developers.

Class	Description
`EventsObservable`	See the source code for a list of all events that Mobile SDK for Android propagates.
`EventsObserver`	Implement this interface to eavesdrop on any event. This functionality is useful if you're doing something special when certain types of events occur.
`UriFragmentParser`	You can directly call this static helper class. It parses a given URI, breaks its parameters into a series of key/value pairs, and returns them in a map.

ForcePlugin Class

All classes in the`com.salesforce.androidsdk.phonegap` package are intended for hybrid app support. Most of these classes implement Javascript plug-ins that access native code. The base class for these Mobile SDK plug-ins is `ForcePlugin`. If you require your own Javascript plug-in in a Mobile SDK app, extend `ForcePlugin`, and implement the abstract `execute()` function.

`ForcePlugin` extends `CordovaPlugin`, which works with the Javascript framework to let you create a Javascript module that can call into native functions. PhoneGap provides the bridge on both sides: you create a native plug-in with `CordovaPlugin` and then you create a Javascript file that mirrors it. Cordova calls the plug-in's `execute()` function when a script calls one of the plug-in's Javascript functions.

Using Passcodes

Passcodes are customer-defined tokens that can provide an extra layer of login security for your app. Optionally, a Salesforce administrator can set the connected app to require a passcode after login. This setting, for example, requires a backgrounded app to prompt for a passcode when it returns to the foreground. When the connected app requires a mobile app passcode, Mobile SDK 9.2 and later use the device system passcode.

To verify a passcode, Mobile SDK presents a lock screen that uses the customer's configured verification mode—for example, biometric, pattern, PIN, or password. If no device passcode has been set, Mobile SDK prompts the customer to create one using any secure input mode supported by the device. If the connected app doesn't require a mobile passcode, Mobile SDK skips the passcode verification step.

Mobile SDK handles all login and passcode lock screens and the authentication handshake. Your app doesn't have to do anything to display these screens.

Mobile Passcode Policies

Each Mobile SDK app hard-codes a connected app's consumer key and OAuth callback URL from a specific Salesforce org. Mobile SDK honors the configurable passcode requirement in that org's designated connected app. Beginning in version 9.2, Mobile SDK ignores org settings such as PIN length, and instead relies on device configuration. Similarly, incorrect passcode entries are handled according to the standard procedure of the mobile operating system.

Note: Beginning in version 9.2, Mobile SDK ignored the **Lock App After** setting in the org's Connected App, in favor of the device's configuration for locking the device after it's been idle. In version 10.1.1 and later, Mobile SDK again respects the **Lock App After** Connected App setting. When set, the mobile app locks after it has been in the background for longer than the timeout period. Locking occurs when the mobile app is activated. Unlocking the app remains the same.

If a customer uses the app to log into a different org, Mobile SDK can't retrieve the designated connected app settings. Therefore, that customer never encounters the passcode prompt.

Multi-User Behavior of the Lock Screen

When multiple users are logged into the same app on the same device, the lock screen behaves as follows.

1. When resuming an app that requires passcode, the customer is first prompted by a lock screen to authenticate through the mobile operating system.
2. If the customer cancels authentication, **Logout** and **Retry Unlock** buttons appear on the lock screen.
3. The **Logout** button works only for customers that require the lock screen.
4. If the last user that requires the lock screen logs out, Mobile SDK no longer shows the lock screen.

Disabling Biometric Identification

Apps built with Mobile SDK 9.2 and later ignore biometric settings from Salesforce connected apps. Instead, customers can configure the authentication mode themselves in device settings.

See Also

- Biometric Authentication on page 420

Resource Handling

In Mobile SDK template apps, resources reside in XML files in the **SalesforceSDK** > **res** project folder. You can customize many of these resources by making changes to these files.

Resources in the `/res` folder are grouped into categories, including:

- Drawables—Backgrounds, drop shadows, image resources such as PNG files
- Layouts—Screen configuration for any visible component, such as the passcode screen
- Menu—Screens for subviews of the login screen, such as the options menu
- Values—Strings, colors, and dimensions that are used by Mobile SDK
- XML—Non-visual configuration settings, such as login server preferences and runtime app restrictions for MDM

Drawable, layout, and value resources are subcategorized into folders that correspond to a variety of form factors. These categories handle different device types and screen resolutions. Each category is defined in its folder name, which allows the resource file name to remain the same for all versions. For example, if the developer provides various sizes of an icon named `icon1.png`, for example, the smart phone version goes in one folder, the low-end phone version goes in another folder, while the tablet icon goes into a third folder. In each folder, the file name is `icon1.png`. The folder names use the same root but with different suffixes.

The following table describes the folder names and suffixes.

Folder name	Usage
`drawable`	Generic versions of drawable resources
`drawable-hdpi`	High resolution; for most smart phones
`drawable-ldpi`	Low resolution; for low-end feature phones
`drawable-mdpi`	Medium resolution; for low-end smart phones
`drawable-xhdpi`	Resources for extra high-density screens (~320dpi
`drawable-xlarge`	For tablet screens in landscape orientation
`drawable-xlarge-port`	For tablet screens in portrait orientation
`drawable-xxhdpi-port`	Resources for extra-extra high density screens (~480 dpi)
`layout`	Generic versions of layouts
`menus`	Add Connection dialog and login menu for phones
`values`	Generic styles and values
`xml`	General app configuration

The compiler looks for a resource in the folder whose name matches the target device configuration. If the requested resource isn't in the expected folder (for example, if the target device is a tablet, but the compiler can't find the requested icon in the `drawables-xlarge` or `drawables-xlarge-port` folder) the compiler looks for the icon file in the generic `drawable` folder.

Layouts

Layouts in the Mobile SDK describe the screen resources that all apps use. For example, layouts configure dialog boxes that handle logins and passcodes.

The name of an XML node in a layout indicates the type of control it describes. For example, the following `TextView` node from `res/layout/sf__passcode.xml` describes a text edit control:

```
<TextView
    android:id="@+id/sf__passcode_title"
    style="@style/SalesforceSDK.Passcode.Text.Title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="17sp"
    android:textColor="@color/sf__passcode_text_color"
    android:textStyle="bold"/>
```

In this case, the `TextView` control is read-only. The style attribute references a global style defined elsewhere in the resources. Instead of specifying style attributes in place, you define styles defined in a central file, and then reference the attribute anywhere it's needed. The value `@style/SalesforceSDK.Passcode.Text.Title` refers to an SDK-owned style defined in `res/values/sf__styles.xml`. Here's the style definition.

```
<style name="SalesforceSDK.Passcode.Text.Title">
    <item name="android:layout_marginTop">@dimen/sf__passcode_title_margin_top</item>
   <item name="android:layout_marginBottom">@dimen/sf__passcode_title_margin_bottom</item>
</style>
```

You can override any style attribute with a reference to one of your own styles. Rather than changing `sf__styles.xml`, define your styles in a different file, such as `xyzcorp__styles.xml`. Place your file in the `res/values` for generic device styles, or the `res/values-xlarge` folder for tablet devices.

Values

The res/values and res/values-xlarge folders contain definitions of style components, such as dimens and colors, string resources, and custom styles. File names in this folder indicate the type of resource or style component.

File name	Contains
`sf__colors.xml`	
sf__attr.xml	Color and integer values used by the Passcode screen
`sf__colors.xml`	Colors referenced by Mobile SDK styles
`sf__dimens.xml`	Dimensions referenced by Mobile SDK styles
`sf__strings.xml`	Strings referenced by Mobile SDK styles; error messages can be overridden
`sf__styles.xml`	Visual styles used by the Mobile SDK
`strings.xml`	App-defined strings

You can override the values in `strings.xml`. To provide your own values, create new files in the same folders using a file name prefix that reflects your own company or project. For example, if your developer prefix is XYZ, you can override `sf__styles.xml` in a new file named `XYZ__styles.xml`.

Other Resources

Two other folders contain Mobile SDK resources.

- `res/menu` defines menus used internally. If your app defines new menus, add them as resources here in new files.
- `res/xml` includes one file that you must edit: `servers.xml`. In this file, change the default Production and Sandbox servers to the login servers for your org. The other files in this folder are for internal use. The `authenticator.xml` file configures the account authentication resource, and the `config.xml` file defines PhoneGap plug-ins for hybrid apps.

SEE ALSO:

Android Resources

Using REST APIs

To query, describe, create, or update data from a Salesforce org, Mobile SDK apps call Salesforce REST APIs. Salesforce REST APIs honor SOQL and SOSL strings and can accept and return data in either JSON or XML format. Mobile SDK wraps standard Salesforce REST requests in methods that handle the low-level HTTP configuration for you. For other Salesforce APIs, Mobile SDK provides methods for manually creating a custom request object and receiving the response. You can even use Mobile SDK REST API methods to make unauthenticated and external API calls.

Salesforce supports an ever-growing variety of REST APIs. For an overview of our offerings, see *Which API Do I Use?* in Salesforce Help. For information on standard REST APIs, see *REST API Developer Guide*.

Coding REST Interactions

With Android native apps, you do minimal coding to access Salesforce data through REST calls. The classes in the `com.salesforce.androidsdk.rest` package initialize the communication channels and encapsulate low-level HTTP plumbing. These classes, all of which are implemented by Mobile SDK, include:

- `ClientManager`—Serves as a factory for `RestClient` instances. It also handles account logins and handshakes with the Salesforce server.
- `RestClient`—Handles protocol for sending REST API requests to the Salesforce server.

  Don't directly create instances of `RestClient`. Instead, call the `ClientManager.getRestClient()` method.
- `RestRequest`—Represents REST API requests formatted from the data you provide. Also serves as a factory for instances of itself.

  Important: Don't directly create instances of `RestRequest`. Instead, call an appropriate `RestRequest` static getter function such as `RestRequest.getRequestForCreate()`.
- `RestResponse`—Contains the response content in the requested format. The `RestRequest` class creates `RestResponse` instances and returns them to your app through your implementation of the `RestClient.AsyncRequestCallback` interface.

Here's the basic procedure for using the REST classes on a UI thread:

1. Create an instance of `ClientManager`.
   a. Use the `SalesforceSDKManager.getInstance().getAccountType()` method to obtain the value to pass as the second argument of the `ClientManager` constructor.
   b. For the `LoginOptions` parameter of the `ClientManager` constructor, call `SalesforceSDKManager.getInstance().getLoginOptions()`.
2. Implement the `ClientManager.RestClientCallback` interface.
3. Call `ClientManager.getRestClient()` to obtain a `RestClient` instance, passing it an instance of your `RestClientCallback` implementation. The following code implements and instantiates `RestClientCallback` inline.

   Kotlin

```
val accountType = SalesforceSDKManager.getInstance().accountType

val loginOptions = SalesforceSDKManager.getInstance().loginOptions
// Get a rest client
ClientManager(this, accountType, loginOptions,
        SalesforceSDKManager.getInstance().shouldLogoutWhenTokenRevoked()).
            getRestClient(this, object : RestClientCallback() {
```

```
    fun authenticatedRestClient(client: RestClient?) {
        if (client == null) {
            SalesforceSDKManager.getInstance().logout(this@MainActivity)
            return
        }
        // Cache the returned client
        this@MainActivity.client = client
    }
}
)
```

Java

```
String accountType =
    SalesforceSDKManager.getInstance().getAccountType();

LoginOptions loginOptions =
    SalesforceSDKManager.getInstance().getLoginOptions();
// Get a rest client
new ClientManager(this, accountType, loginOptions,
   SalesforceSDKManager.getInstance().
   shouldLogoutWhenTokenRevoked()).
   getRestClient(this, new RestClientCallback() {
      @Override
      public void
      authenticatedRestClient(RestClient client) {
         if (client == null) {
            SalesforceSDKManager.getInstance().
               logout(MyActivity.this);
            return;
         }
         // Cache the returned client
         MyActivity.this.client = client;
      }
   }
);
```

4. Call a static `RestRequest()` getter method to obtain the appropriate `RestRequest` object for your needs. For example, to get a description of a Salesforce object:

   ```
   final RestRequest request = RestRequest.getRequestForDescribe(apiVersion, objectType);
   ```

5. Pass the `RestRequest` object you obtained in the previous step to `RestClient.sendAsync()` or `RestClient.sendSync()`. If you're on a UI thread and therefore calling `sendAsync()`:
   a. Implement the `ClientManager.AsyncRequestCallback` interface.
   b. Pass an instance of your implementation to the `sendAsync()` method.
   c. Receive the formatted response through your `ASyncRequestCallback.onSuccess()` method. Before using the response, double-check that it's valid by calling `RestResponse.isSuccess()`.

The following code implements and instantiates `AsyncRequestCallback` inline.

Kotlin

```
private fun sendFromUIThread(restRequest: RestRequest) {
    client.sendAsync(restRequest, object : AsyncRequestCallback {
        private val start = System.nanoTime()
        override fun onSuccess(request: RestRequest, result: RestResponse) {
            // Consume before going back to main thread
            // Not required if you don't do main (UI) thread tasks here
            result.consumeQuietly()
            runOnUiThread {
                // Network component doesn’t report app layer status.
                // Use the Mobile SDK RestResponse.isSuccess() method to check
                // whether the REST request itself succeeded.
                if (result.isSuccess) {
                    try {
                        // Do something with the result
                    } catch (e: Exception) {
                        printException(e)
                    }

                    EventsObservable.get().notifyEvent(EventType.RenditionComplete)
                }
            }
        }

        override fun onError(exception: Exception) {
            printException(exception)
            EventsObservable.get().notifyEvent(EventType.RenditionComplete)
        }
    })
}
```

Java

```
private void sendFromUIThread(RestRequest restRequest) {
    client.sendAsync(restRequest, new AsyncRequestCallback() {
        private long start = System.nanoTime();
        @Override
        public void onSuccess(RestRequest request, final RestResponse result) {
         // Consume before going back to main thread
         // Not required if you don't do main (UI) thread tasks here
         result.consumeQuietly();
         runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    // Network component doesn’t report app layer status.
                    // Use the Mobile SDK RestResponse.isSuccess() method to check
                    // whether the REST request itself succeeded.
                    if (result.isSuccess()) {
                        try {
                            // Do something with the result
                        }
                        catch (Exception e) {
                            printException(e);
                        }
```

```
                        EventsObservable.get().notifyEvent(EventType.RenditionComplete);

                    }
                }
            });
        }
        @Override
        public void onError(Exception exception)
        {
            printException(exception);
            EventsObservable.get().notifyEvent(EventType.RenditionComplete);
        }
    });
}
```

If you're calling the `sendSync()` method from a service, use the same procedure with the following changes.

1. To obtain a `RestClient` instance call `ClientManager.peekRestClient()` instead of `ClientManager.getRestClient()`.
2. Retrieve your formatted REST response from the `sendSync()` method's return value.

Checking REST Response Status

A REST response arriving at your app's onSuccess() callback method indicates only that the network call didn't fail. This high-level status doesn't factor in app-level success or failure.

In Mobile SDK for Android, the `RestResponse` object wraps the underlying `okHttp3.Response`. To help you code more defensively, `RestResponse` provides the following convenience methods for inspecting response details.

public isSuccess()

public static boolean isSuccess(int statusCode)
Returns true if the HTTP response status code or the given code is between 200 and 299, indicating app-level success.

public int getStatusCode()
Returns the response status code.

public String getContentType()
Returns the `content-type` header, if found.

public Map<String, List<String>> getAllHeaders()
Returns all headers associated with this response.

public Response getRawResponse()
Returns the underlying `okHttp3.Response` object.

Batch and Composite Requests

Batch and composite APIs pose special challenges, because they handle multiple requests in a single call. Mobile SDK classes take the pain out of building and configuring these complex requests.

Batch and Composite Request Classes

- Request:

```
BatchRequest
CompositeRequest
```

- Builder:

```
BatchRequestBuilder
CompositeRequestBuilder
```

- Response:

```
BatchResponse
CompositeRequestBuilder
```

These classes make it easy to create batch and composite requests. To use them:

1. Create a builder instance. For batch requests, you can optionally set `haltOnError` property to true:

   Kotlin

   ```
   val builder = BatchRequest.BatchRequestBuilder()
   // Optional; defaults to false
   builder.setHaltOnError(true)
   ```

   Java

   ```
   BatchRequest.BatchRequestBuilder builder =
       new BatchRequest.BatchRequestBuilder();
   // Optional; defaults to false
   builder.setHaltOnError(true);
   ```

   For composite requests, you can optionally set the `allOrNone` rollback property to true.

   Kotlin

   ```
   val builder = CompositeRequest.CompositeRequestBuilder()
   // Optional; defaults to false
   builder.setAllOrNone(true)
   ```

   Java

   ```
   CompositeRequest.CompositeRequestBuilder builder =
       new CompositeRequest.CompositeRequestBuilder();
   // Optional; defaults to false
   builder.setAllOrNone(true);
   ```

2. As you create REST requests, add them to the builder object. For batch requests:

   Kotlin

   ```
   builder.addRequest(request)
   ```

   Java

   ```
   builder.addRequest(request);
   ```

With composite requests, you also provide a reference ID as described in the *REST API Developer Guide*. You specify this ID when you add the `RestRequest` object:

Kotlin

```
builder.addRequest(referenceId, request)
```

Java

```
builder.addRequest(referenceId, request);
```

3. When you're ready, call the builder object's build method:

   Kotlin

   ```
   builder.build(ApiVersionStrings.VERSION_NUMBER)
   ```

   Java

   ```
   builder.build(ApiVersionStrings.VERSION_NUMBER);
   ```

   Each build method returns a specialized `RestRequest` object (`BatchRestRequest` or `CompositeRestRequest` instance) that you can send through the shared `RestClient.sendAsync()` method.

   Tip: To use an older API version as the default argument, pass a literal string in the format "v42.0".

4. Responses to batch and composite requests arrive as instances of the response class (`BatchResponse` or `CompositeResponse`).

See Also

- "Batch" in *REST API Developer Guide*
- "Composite" in *REST API Developer Guide*

Unauthenticated REST Requests

In certain cases, some applications must make REST calls before the user becomes authenticated. In other cases, the application must access services outside of Salesforce that don't require Salesforce authentication. To implement such requirements, use a special `RestClient` instance that doesn't require an authentication token.

To obtain an unauthenticated `RestClient` on Android, use one of the following `ClientManager` factory methods:

```
/**
* Method to create an unauthenticated RestClient asynchronously
* @param activityContext
* @param restClientCallback
*/
public void getUnauthenticatedRestClient(Activity activityContext, RestClientCallback
restClientCallback);
```

```
/**
* Method to create an unauthenticated RestClient.
* @return
*/
public RestClient peekUnauthenticatedRestClient();
```

Note: A REST request sent through either of these `RestClient` objects requires a full path URL. Mobile SDK doesn't prepend an instance URL to unauthenticated endpoints.

Example:

Kotlin

```
val unauthenticatedRestClient = clientManager.peekUnauthenticatedRestClient()
val request = RestRequest(RestMethod.GET,
    "https://api.spotify.com/v1/search?q=James%20Brown&type=artist", null)
val response = unauthenticatedRestClient.sendSync(request)
```

Java

```
RestClient unauthenticatedRestClient = clientManager.peekUnauthenticatedRestClient();
RestRequest request = new RestRequest(RestMethod.GET,
    "https://api.spotify.com/v1/search?q=James%20Brown&type=artist", null);

RestResponse response = unauthenticatedRestClient.sendSync(request);
```

Deferring Login in Native Android Apps

When you create Mobile SDK apps using forcedroid, forcedroid bases your project on a template app that gives you lots of free standard functionality. For example, you don't have to implement authentication—login and passcode handling are built into your launcher activity. This design works well for most apps, and the free code is a big time-saver. However, after you've created your forcedroid app you might find reasons for deferring Salesforce authentication until some point after the launcher activity runs.

You can implement deferred authentication easily while keeping the template app's built-in functionality. Here are the guidelines and caveats:

- Replace the launcher activity (named `MainActivity` in the template app) with an activity that does *not* extend any of the following Mobile SDK activities:
  - `SalesforceActivity`
  - `SalesforceListActivity`
  - `SalesforceExpandableListActivity`

  This rule likewise applies to any other activities that run before you authenticate with Salesforce.
- Do not call the `peekRestClient()` or the `getRestClient()` `ClientManager` method from your launcher activity or from any other pre-authentication activities.
- Do not change the `initNative()` call in the `TemplateApp` class. It must point to the activity class that launches after authentication (`MainActivity` in the template app).
- When you're ready to authenticate with Salesforce, launch the `MainActivity` class.

The following example shows how to place a non-Salesforce activity ahead of Salesforce authentication. You can of course expand and embellish this example with additional pre-authentication activities, observing the preceding guidelines and caveats. This example is based on the MobileSyncExplorer sample app.

1. Create an XML layout for the pre-authentication landing page of your application. For example, the following layout file, `launcher.xml`, contains only a button that triggers the login flow.

Note: The following example defines a string resource, `@string/login`, in the `res/strings.xml` file as follows:

```
<string name="login">Login</string>
```

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="@android:color/white"
    android:id="@+id/root">

    <Button android:id="@+id/login_button"
        android:layout_width="80dp"
        android:layout_height="60dp"
        android:text="@string/login"
        android:textColor="@android:color/black"
        android:textStyle="bold"
        android:gravity="center"
        android:layout_gravity="center"
        android:textSize="18sp"
        android:onClick="onLoginClicked" />
</LinearLayout>
```

2. Create a landing screen activity. For example, here's a landing screen activity named `LauncherActivity`. This screen simply inflates the XML layout defined in `launcher.xml`. This class must not extend any of the Salesforce activities or call `peekRestClient()` or `getRestClient()`, since these calls trigger the authentication flow. When the user taps the login button, the `onLoginClicked()` button handler launches `MainActivity`, and login ensues.

   Kotlin

```
class LauncherActivity : Activity() {
    public override fun onCreate(savedInstance: Bundle?) {
        super.onCreate(savedInstance)
        setContentView(R.layout.launcher)
    }

    /**
     * Callback received when the 'Delete' button is clicked.
     *
     * @param v View that was clicked.
     */
    fun onLoginClicked(v: View) {
        /*
         * TODO: Add logic here to determine if we are already
         * logged in, and skip this screen by calling
         * 'finish()', if that is the case.
         */
        val mainIntent = Intent(this, MainActivity::class.java)
        mainIntent.addCategory(Intent.CATEGORY_DEFAULT)
        startActivity(mainIntent)
        finish()
```

```
    }
}
```

Java

```
package com.salesforce.samples.mobilesyncexplorer.ui;

import com.salesforce.samples.mobilesyncexplorer.R;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class LauncherActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstance) {
        super.onCreate(savedInstance);
        setContentView(R.layout.launcher);
    }

    /**
      * Callback received when the 'Delete' button is clicked.
      *
      * @param v View that was clicked.
    */
    public void onLoginClicked(View v) {
        /*
         * TODO: Add logic here to determine if we are already
         * logged in, and skip this screen by calling
         * 'finish()', if that is the case.
        */
        final Intent mainIntent =
            new Intent(this, MainActivity.class);
        mainIntent.addCategory(Intent.CATEGORY_DEFAULT);
        startActivity(mainIntent);
        finish();
    }
}
```

3. Modify the `AndroidManifest.xml` to specify `LauncherActivity` as the activity to be launched when the app first starts.

```
<!-- Launcher screen -->
<activity android:name=
"com.salesforce.samples.mobilesyncexplorer.ui.LauncherActivity"
   android:label="@string/app_name"
   android:theme="@style/SalesforceSDK.ActionBarTheme">
   <intent-filter>
       <action android:name="android.intent.action.MAIN" />
       <category
           android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
</activity>
```

```
<!-- Main screen -->
<activity android:name=
"com.salesforce.samples.mobilesyncexplorer.ui.MainActivity"
   android:label="@string/app_name"
   android:theme="@style/SalesforceSDK.ActionBarTheme">
   <intent-filter>
       <category android:name=
           "android.intent.category.DEFAULT" />
   </intent-filter>
</activity>
```

When you start the application. the `LauncherActivity` screen appears. Click the login button to initiate the Salesforce authentication flow. After authentication completes, the app launches `MainActivity`.

Android Template App: Deep Dive

To create new Android apps, the `forcedroid create` command repurposes one of two Mobile SDK template projects.

- **AndroidNativeTemplate**—Implements a basic Mobile SDK native app using Java.
- **AndroidNativeTemplateKotlin**—Implements a basic Mobile SDK native app using Kotlin.

You can find both projects in the github.com/forcedotcom/SalesforceMobileSDK-Templates GitHub repo.

By studying a template app, you can gain a quick understanding of native apps built with Mobile SDK for Android.

Template projects define two classes: `MainApplication` and `MainActivity`.

- The `MainApplication` class extends the Android `Application` class and calls `MobileSyncSDKManager.initNative()` in its `onCreate()` override.
- The `MainActivity` class extends the `SalesforceActivity` class.

These two classes create a running mobile app that displays a login screen and a home screen.

Despite containing only about 200 lines of code, the Mobile SDK template apps are more than just "Hello World" examples. In its main activity, it retrieves Salesforce data through REST requests and displays the results on a mobile page. You can extend these apps by adding more activities, calling other components, and doing anything else that the Android operating system, the device, and your security constraints allow.

MainApplication Class

Every native Android app requires an instance of `android.app.Application`. The `MainApplication` class accomplishes these basic tasks:

- Overrides the Android `Application.onCreate()` method.
- In its `onCreate()` override:
  - Calls the superclass `onCreate()` method.
  - Initializes Salesforce Mobile SDK by calling `initNative()` on the SDK manager object (`MobileSyncSDKManager`).
  - Provides optional commented code that you can reinstate to use your app as a Salesforce identity provider.
  - Provides optional commented code that you can reinstate to support push notifications.

Here's the entire class:

Kotlin

```
package com.bestapps.android

import android.app.Application
import com.salesforce.androidsdk.mobilesync.app.MobileSyncSDKManager

/**
 * Application class for our application.
 */
class MainApplication : Application() {

    companion object {
        private const val FEATURE_APP_USES_KOTLIN = "KT"
    }

    override fun onCreate() {
        super.onCreate()
        MobileSyncSDKManager.initNative(applicationContext, MainActivity::class.java)
       MobileSyncSDKManager.getInstance().registerUsedAppFeature(FEATURE_APP_USES_KOTLIN)

        /*
         * Uncomment the following line to enable IDP login flow. This will allow the
user to
         * either authenticate using the current app or use a designated IDP app for
login.
         * Replace 'idpAppURIScheme' with the URI scheme of the IDP app meant to be
used.
         */
        // MobileSyncSDKManager.getInstance().idpAppURIScheme = idpAppURIScheme

        /*
   * Un-comment the line below to enable push notifications in this app.
   * Replace 'pnInterface' with your implementation of 'PushNotificationInterface'.
   * Add your Google package ID in 'bootonfig.xml', as the value
   * for the key 'androidPushNotificationClientId'.
   */
        // MobileSyncSDKManager.getInstance().pushNotificationReceiver = pnInterface
    }
}
```

Java

```
package com.salesforce.androidnativetemplate;
import android.app.Application;
import com.salesforce.androidsdk.mobilesync.app.MobileSyncSDKManager;
/**
 * Application class for our application.
 */
public class MainApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        MobileSyncSDKManager.initNative(getApplicationContext(), MainActivity.class);
```

```
        /*
         * Uncomment the following line to enable IDP login flow. This will allow the
user to
         * either authenticate using the current app or use a designated IDP app for
login.
         * Replace 'idpAppURIScheme' with the URI scheme of the IDP app meant to be
used.
         */
        // MobileSyncSDKManager.getInstance().setIDPAppURIScheme(idpAppURIScheme);

        /*
         * Un-comment the line below to enable push notifications in this app.
        * Replace 'pnInterface' with your implementation of 'PushNotificationInterface'.

         * Add your Google package ID in 'bootonfig.xml', as the value
         * for the key 'androidPushNotificationClientId'.
         */
       // MobileSyncSDKManager.getInstance().setPushNotificationReceiver(pnInterface);

    }
}
```

Most native Android apps can use similar code. For this small amount of work, your app gets free implementations of passcode and login/logout mechanisms, plus a few other benefits. See SalesforceActivity, SalesforceListActivity, and SalesforceExpandableListActivity Classes.

MainActivity Class

In Mobile SDK apps, the main activity begins immediately after the user logs in. Once the main activity is running, it can launch other activities, which in turn can launch sub-activities. When the application exits, it does so by terminating the main activity. All other activities terminate in a cascade from within the main activity.

The template app's `MainActivity` class extends the abstract Mobile SDK activity class, `com.salesforce.androidsdk.ui.SalesforceActivity`. This superclass gives you free implementations of mandatory passcode and login protocols. If you use another base activity class instead, you're responsible for implementing those protocols. `MainActivity` initializes the app's UI and implements its UI buttons.

The `MainActivity` UI includes a list view that can show the user's Salesforce Contacts or Accounts. When the user clicks one of these buttons, the `MainActivity` object performs a couple of basic queries to populate the view. For example, to fetch the user's Contacts from Salesforce, the `onFetchContactsClick()` message handler sends a simple SOQL query:

Kotlin

```
@Throws(UnsupportedEncodingException::class)
@Suppress("UNUSED_PARAMETER")
fun onFetchContactsClick(v: View) {
    sendRequest("SELECT Name FROM Contact")
}
```

Java

```
public void onFetchContactsClick(View v) throws UnsupportedEncodingException {
    sendRequest("SELECT Name FROM Contact");
}
```

Internally, the private `sendRequest()` method formulates a server request using the `RestRequest` class and the given SOQL string:

Kotlin

```
@Throws(UnsupportedEncodingException::class)
private fun sendRequest(soql: String) {
    val restRequest =
RestRequest.getRequestForQuery(ApiVersionStrings.getVersionNumber(this), soql)

    client!!.sendAsync(restRequest, object : AsyncRequestCallback {
        override fun onSuccess(request: RestRequest, result: RestResponse) {
            result.consumeQuietly() // consume before going back to main thread
            runOnUiThread {
                try {
                    listAdapter!!.clear()
                    val records = result.asJSONObject().getJSONArray("records")
                    for (i in 0..records.length() - 1) {
                        listAdapter!!.add(records.getJSONObject(i).getString("Name"))
                    }
                } catch (e: Exception) {
                    onError(e)
                }
            }
        }

        override fun onError(exception: Exception) {
            runOnUiThread {
                Toast.makeText(this@MainActivity,
                        this@MainActivity.getString(R.string.sf__generic_error,
exception.toString()),
                        Toast.LENGTH_LONG).show()
            }
        }
    })
}
```

Java

```
private void sendRequest(String soql) throws UnsupportedEncodingException
{
    RestRequest restRequest =
        RestRequest.getRequestForQuery(
            getString(R.string.api_version), soql);
    client.sendAsync(restRequest, new AsyncRequestCallback()
    {
        @Override
        public void onSuccess(RestRequest request,
            RestResponse result) {
            // Consume before going back to main thread
            result.consumeQuietly();
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    // Network component doesn't report app layer status.
                    // Use the Mobile SDK RestResponse.isSuccess() method to check
```

```
                    // whether the REST request itself succeeded.
                     if (result.isSuccess()) {
                         try {
                             listAdapter.clear();
                       JSONArray records =
                           result.asJSONObject().getJSONArray("records");
                             for (int i = 0; i < records.length(); i++) {
                                 listAdapter.add(
                                     records.getJSONObject(i).getString("Name"));
                             }
                         } catch (Exception e) {
                             onError(e);
            }
        }
                }
            });
        }
        @Override
        public void onError(final Exception exception) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Toast.makeText(MainActivity.this,
                        MainActivity.this.getString(R.string.sf__generic_error,
exception.toString()),
                        Toast.LENGTH_LONG).show();
                }
            });
        }
    });
}
```

This method uses an instance of the `com.salesforce.androidsdk.rest.RestClient` class, `client`, to process its SOQL query. The `RestClient` class relies on two helper classes—`RestRequest` and `RestResponse`—to send the query and process its result. The `sendRequest()` method calls `RestClient.sendAsync()` to process the SOQL query asynchronously.

To support the `sendAsync()` call, the `sendRequest()` method constructs an instance of `com.salesforce.androidsdk.rest.RestRequest`, passing it the API version and the SOQL query string. The resulting object is the first argument for `sendAsync()`. The second argument is a callback object. When `sendAsync()` has finished running the query, it sends the results to this callback object. If the query is successful, the callback object uses the query results to populate a UI list control. If the query fails, the callback object displays a toast popup to display the error message.

Overriding the Default API Version

For Android, Mobile SDK core libraries define a hard-coded default API version. This value matches the Salesforce API version on the date of the current Mobile SDK release. You can override the default version in your app by setting `api_version` in the `strings.xml` resource file.

If you override the default version, be careful when calling methods such as `RestRequest.getRequestForQuery()` that require an API version argument. It's tempting to pass `ApiVersionStrings.getVersionNumber()`, but in some cases this call returns unexpected values. Here are some tips.

- To return either your overridden version or the default value if no override exists, set the `context` argument to a valid subclass of `Context`. High-level subclasses include extensions of `Activity`, `Service`, and `Application`.

- If you're calling `getVersionNumber()` from a class that isn't a `Context` subclass, you can pass `SalesforceSDKManager.getInstance().getAppContext()` as the `context` argument.
- If you set the `context` argument to null, `getVersionNumber()` always returns the hard-coded default value.

Using an Anonymous Class in Java

In the call to `RestClient.sendAsync()` the code instantiates a new `AsyncRequestCallback` object as its second argument. However, the `AsyncRequestCallbackconstructor` is followed by a code block that overrides a couple of methods: `onSuccess()` and `onError()`. If that code looks strange to you, take a moment to see what's happening. `AsyncRequestCallback` is defined as an interface, so it has no implementation. In order to instantiate it, the code implements the two `AsyncRequestCallback` methods inline to create an anonymous class object. This technique gives `TemplateApp` a `sendAsync()` implementation of its own that can never be called from another object and doesn't litter the API landscape with a group of specialized class names.

Template Manifest

A look at the `AndroidManifest.xml` file in the template project reveals the components required for Mobile SDK native Android apps. The only required component is the activity named "`.MainActivity`". This component represents the first activity that is called after login. A class by this name is defined in the project.

Because any app created by forcedroid is based on a Mobile SDK template project, the `MainActivity` component is already included in its manifest. As with any Android app, you can add other components, such as custom activities or services, by editing the manifest in Android Studio.

Android Sample Applications

Salesforce Mobile SDK includes the following native Android sample applications.

- **MobileSyncExplorer** demonstrates the power of the native Mobile Sync library on Android. It resides in Mobile SDK for Android under `native/NativeSampleApps/MobileSyncExplorer`.
- **RestExplorer** demonstrates the OAuth and REST API functions of Mobile SDK. It's also useful for investigating REST API actions from a tablet.

Mobile SDK also provides Android wrappers for a few hybrid apps under `hybrid/HybridSampleApps/`.

- **AccountEditor**: Demonstrates how to synchronize offline data using the `mobilesync.js` library.
- **MobileSyncExplorerHybrid**: Demonstrates how to synchronize offline data using the Mobile Sync plugin.
- **NoteSync**: Demonstrates how to use non-REST APIs to retrieve Salesforce Notes.

CHAPTER 8 HTML5 and Hybrid Development

In this chapter ...

HTML5 lets you create lightweight mobile interfaces without installing software on the target device. Any mobile, touch or desktop device can access these mobile interfaces. HTML5 now supports advanced mobile functionality such as camera and GPS, making it simple to use these popular device features in your Salesforce mobile app.

You can create an HTML5 application that leverages the Salesforce Platform by:

- Using Visualforce to deliver the HTML content
- Using JavaScript remoting to invoke Apex controllers for fetching records from Salesforce

In addition, you can repurpose HTML5 code in a standalone Mobile SDK hybrid app, and then distribute it through an app store. To convert to hybrid, you use the third-party Cordova command line to create a Mobile SDK container project, and then import your HTML5, JavaScript, and CSS files into that project.

Getting Started

If you're already a web developer, you're set up to write HTML5 apps that access Salesforce. HTML5 apps can run in a browser and don't require the Salesforce Mobile SDK. You simply call Salesforce APIs, capture the return values, and plug them into your logic and UI. The same advantages and challenges of running any app in a mobile browser apply. However, Salesforce and its partners provide tools that help streamline mobile web design and coding.

If you want to build your HTML5 app as standalone in a hybrid container and distribute it in the Apple App Store or an Android marketplace, you'll need to create a hybrid app using the Mobile SDK.

Using HTML5 and JavaScript

You don't need a professional development environment such as Xcode or Microsoft® Visual Studio® to write HTML5 and JavaScript code. Most modern browsers include sophisticated developer features including HTML and JavaScript debuggers. You can literally write your application in a text editor and test it in a browser. However, you do need a good knowledge of popular industry libraries that can help to minimize your coding effort.

The recent growth in mobile development has led to an explosion of new web technology toolkits. Often, these JavaScript libraries are open-source and don't require licensing. Most of the tools provided by Salesforce for HTML5 development are built on these third-party technologies.

HTML5 Development Requirements

If you're planning to write a browser-based HTML5 Salesforce application, you don't need Salesforce Mobile SDK.

- You'll need a Salesforce org.
- Some knowledge of Apex and Visualforce is necessary.

Note: This type of development uses Visualforce. You can't use Database.com.

Multi-Device Strategy

With the worldwide proliferation of mobile devices, HTML5 mobile applications must support a variety of platforms, form factors, and device capabilities. Developers who write device-independent mobile apps in Visualforce face these key design questions:

- Which devices and form factors should my app support?
- How does my app detect various types of devices?
- How should I design a Salesforce application to best support multiple device types?

Which Devices and Form Factors Should Your App Support?

The answer to this question is dependent on your specific use case and end-user requirements. It is, however, important to spend some time thinking about exactly which devices, platforms, and form factors you do need to support. Where you end up in the spectrum of 'Support all platforms/devices/form factors' to 'Support only desktop and iPhone' (as an example) plays a major role in how you answer the subsequent two questions.

As can be expected, important trade-offs have to be made when making this decision. Supporting multiple form factors obviously increases the reach for your application. But, it comes at the cost of additional complexity both in terms of initially developing the application, and maintaining it over the long-term.

Developing true cross-device applications is not simply a question of making your web page look (and perform) optimally across different form factors and devices (desktop vs phone vs tablet). You really need to rethink and customize the user experience for each specific device/form factor. The phone or tablet version of your application very often does not need all the bells and whistles supported by your existing desktop-optimized Web page (e.g., uploading files or supporting a use case that requires many distinct clicks).

Conversely, the phone/tablet version of your application can support features like geolocation and taking pictures that are not possible in a desktop environment. There are even significant differences between the phone and tablet versions of the better designed applications like LinkedIn and Flipboard (e.g,. horizontal navigation in a tablet version vs single hand vertical scrolling for a phone version). Think of all these consideration and the associated time and cost it will take you to support them when deciding which devices and form factors to support for your application.

Once you've decided which devices to support, you then have to detect which device a particular user is accessing your Web application from.

Client-Side Detection

The client-side detection approach uses JavaScript (or CSS media queries) running on the client browser to determine the device type. Specifically, you can detect the device type in two different ways.

- **Client-Side Device Detection with the User-Agent Header** — This approach uses JavaScript to parse out the User-Agent HTTP header and determine the device type based on this information. You could of course write your own JavaScript to do this. A better option is to reuse an existing JavaScript. A cursory search of the Internet will result in many reusable JavaScript snippets that can detect the device type based on the User-Agent header. The same cursory search, however, will also expose you to some of the perils of using this approach. The list of all possible User-Agents is huge and ever growing and this is generally considered to be a relatively unreliable method of device detection.
- **Client-Side Device Detection with Screen Size and/or Device Features** — A better alternative to sniffing User-Agent strings in JavaScript is to determine the device type based on the device screen size and or features (e.g., touch enabled). One example of this approach can be found in the open-source Contact Viewer HTML5 mobile app that is built entirely in Visualforce. Specifically, the MobileAppTemplate.page includes a simple JavaScript snippet at the top of the page to distinguish between phone and tablet clients based on the screen size of the device. Another option is to use a library like Device.js or Modernizr to detect the device type. These libraries use some combination of CSS media queries and feature detection (e.g., touch enabled) and are therefore a more reliable option for detecting device type. A simple example that uses the Modernizr library to accomplish this can be found at `http://www.html5rocks.com/static/demos/cross-device/feature/index.html`. A more complete example that uses the `Device.js` library and integrates with Visualforce can be found in this GitHub repo: `https://github.com/sbhanot-sfdc/Visualforce-Device.js`. Here is a snippet from the DesktopVersion.page in that repo.

```
<apex:page docType="html-5.0" sidebar="false" showHeader="false" standardStylesheets="false"
 cache="false" >

<head>
  <!-- Every version of your webapp should include a list of all
  versions. -->
  <link rel="alternate" href="/apex/DesktopVersion" id="desktop"
    media="only screen and (touch-enabled: 0)"/>
  <link rel="alternate" href="/apex/PhoneVersion" id="phone"
    media="only screen and (max-device-width: 640px)"/>
  <link rel="alternate" href="/apex/TabletVersion" id="tablet"
    media="only screen and (min-device-width: 641px)"/>

  <meta name="viewport" content="width=device-width, user-scalable=no"/>
  <script src="{!URLFOR($Resource.Device_js)}"/>
</head>
```

```
<body>
  <ul>
    <li><a href="?device=phone">Phone Version</a></li>
    <li><a href="?device=tablet">Tablet Version</a></li>
  </ul>
  <h1> This is the Desktop Version</h1>
</body>
</apex:page>
```

The snippet above shows how you can simply include a `<link>` tag for each device type that your application supports. The `Device.js` library then automatically redirects users to the appropriate Visualforce page based on device type detected. There is also a way to override the default Device.js redirect by using the '?device=xxx' format shown above.

Server-Side Device Detection

Another option is to detect the device type on the server (i.e., in your Apex controller/extension class). Server-side device detection is based on parsing the User-Agent HTTP header and here is a small code snippet of how you can detect if a Visualforce page is being viewed from an iPhone client.

```
<apex:page docType="html-5.0"
    sidebar="false"
    showHeader="false"
    cache="false"
    standardStylesheets="false"
    controller="ServerSideDeviceDetection"
    action="{!detectDevice}">
  <h1> This is the Desktop Version</h1>
</apex:page>
```

```
public with sharing class ServerSideDeviceDetection {
    public boolean isIPhone {get;set;}
    public ServerSideDeviceDetection() {
         String userAgent =
                System.currentPageReference().
                    getHeaders().get('User-Agent');
         isIPhone = userAgent.contains('iPhone');
    }
    public PageReference detectDevice(){
        if (isIPhone)
            return Page.PhoneVersion.setRedirect(true);
        else
            return null;
    }
}
```

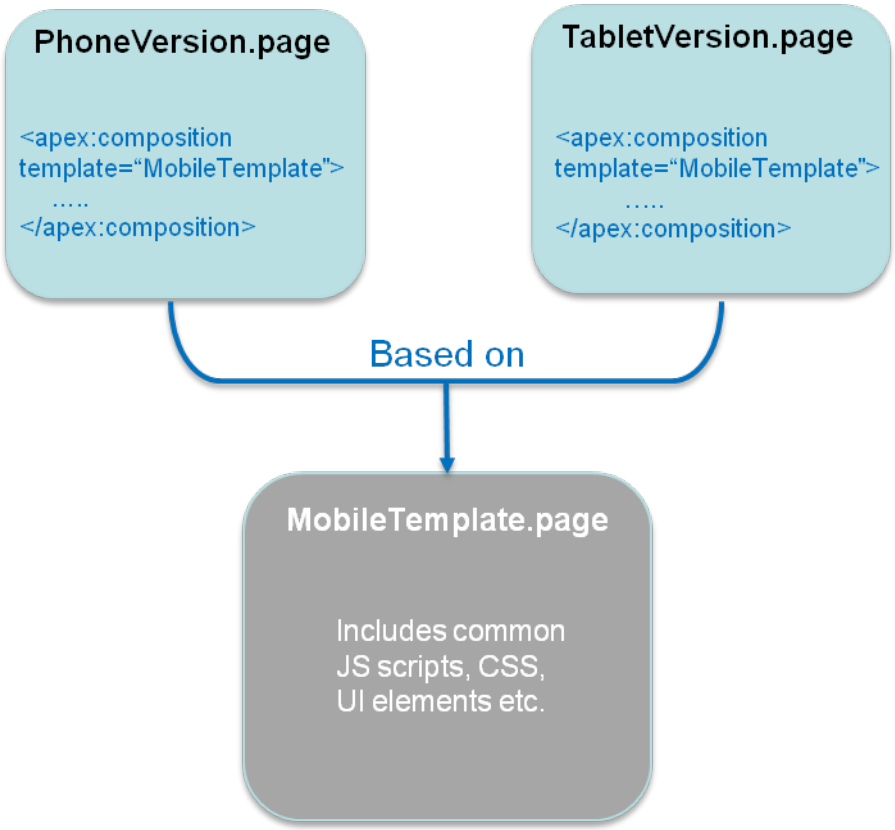
Note that User-Agent parsing in the code snippet above is far from comprehensive and you should implement something more robust that detects all the devices that you need to support based on regular expression matching. A good place to start is to look at the RegEx included in the detectmobilebrowsers.com code snippets.

How Should You Design a Salesforce Application to Best Support Multiple Device Types?

Finally, once you know which devices you need to support and how to distinguish between them, what is the optimal application design for delivering a customized user experiences for each device/form factor? Again, a couple of options to consider.

For simple applications where all you need is for the same Visualforce page to display well across different form factors, a responsive design approach is an attractive option. In a nutshell, Responsive design uses CCS3 media queries to dynamically reformat a page to fit the form factor of the client browser. You could even use a responsive design framework like Twitter Bootstrap to achieve this.

Another option is to design multiple Visualforce pages, each optimized for a specific form factor and then redirect users to the appropriate page using one of the strategies described in the previous section. Note that having separate Visualforce pages does not, and should not, imply code/functionality duplication. A well architected solution can maximize code reuse both on the client-side (by using Visualforce strategies like Components, Templates etc.) as well as the server-side (e.g., encapsulating common business logic in an Apex class that gets called by multiple page controllers). An excellent example of such a design can be found in the same open-source Contact Viewer application referenced before. Though the application has separate pages for its phone and tablet version (`ContactsAppMobile.page` and `ContactsApp.page` respectively), they both share a common template (`MobileAppTemplate.page`), thus maximizing code and artifact reuse. The figure below is a conceptual representation of the design for the Contact Viewer application.



Lastly, it is also possible to service multiple form factors from a single Visualforce page by doing server-side device detection and making use of the 'rendered' attribute available in most Visualforce components (or more directly, the CSS 'display:none/block' property on a tag) to selectively show/hide page elements. This approach however can result in bloated and hard-to-maintain code and should be used sparingly.

Delivering HTML5 Content With Visualforce

Traditionally, you use Visualforce to create custom websites for the desktop environment. When combined with HTML5, however, Visualforce becomes a viable delivery mechanism for mobile Web apps. These apps can leverage third-party UI widget libraries such as Sencha, or templating frameworks such as AngularJS and Backbone.js, that bind to data inside Salesforce.

To set up an HTML5 Apex page, change the `docType` attribute to "html-5.0", and use other settings similar to these:

```
<apex:page docType="html-5.0" sidebar="false" showHeader="false" standardStylesheets="false"
 cache="true" >

</apex:page>
```

This code sets up an Apex page that can contain HTML5 content, but, of course, it produces an empty page. With the use of static resources and third-party libraries, you can add HTML and JavaScript code to build a fully interactive mobile app.

Accessing Salesforce Data: Controllers vs. APIs

In an HTML5 app, you can access Salesforce data two ways.

- By using JavaScript remoting to invoke your Apex controller.
- By accessing the Salesforce API with `force.js`.

Using JavaScript Remoting to Invoke Your Apex Controller

Apex supports the following two means of invoking Apex controller methods from JavaScript:

- `apex:actionFunction`
- JavaScript remoting

Both techniques use an AJAX request to invoke Apex controller methods directly from JavaScript. The JavaScript code must be hosted on a Visualforce page.

In comparison to `apex:actionFunction`, JavaScript remoting offers several advantages.

- It offers greater flexibility and better performance than `apex:actionFunction`.
- It supports parameters and return types in the Apex controller method, with automatic mapping between Apex and JavaScript types.
- It uses an asynchronous processing model with callbacks.
- Unlike `apex:actionFunction`, the AJAX request does not include the view state for the Visualforce page. This results in a faster round trip.

Compared to `apex:actionFunction`, however, JavaScript remoting requires you to write more code.

The following example inserts JavaScript code in a <script> tag on the Visualforce page. This code calls the `invokeAction()` method on the Visualforce remoting manager object. It passes `invokeAction()` the metadata needed to call a function named `getItemId()` on the Apex controller object `objName`. Because `invokeAction()` runs asynchronously, the code also defines a callback function to process the value returned from `getItemId()`. In the Apex controller, the `@RemoteAction` annotation exposes the `getItemId()` function to external JavaScript code.

```
//Visualforce page code
<script type="text/javascript">
```

```
     Visualforce.remoting.Manager.invokeAction(
        '{!$RemoteAction.MyController.getItemId}',
        objName,
        function(result, event){
            //process response here
        },
        {escape: true}
    );
<script>

//Apex Controller code

@RemoteAction
global static String getItemId(String objectName) { ... }
```

See https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_annotation_RemoteAction.htm to learn more about `@RemoteAction` annotations.

Accessing the Salesforce API with Force.js

The following sample code queries Salesforce records from Apex by using the the `cordova.js` and `force.js` libraries. To add these resources to your Apex page:

1. Create an archive file, such as a ZIP file, that contains `cordova.js`, `force.js`, and any other static resources your project requires.
2. In Salesforce, upload the archive file via **Your Name** > **App Setup** > **Develop** > **Static Resources**.

The sample code uses an instance of the `force.js` library to log in to Salesforce. It then calls the `force.query()` method to process a SOQL query. The query callback function displays the `Name` fields returned by the query as HTML in an object with ID "contacts". At the end of the Apex page, the HTML5 content defines the `contacts` element as a simple `<ul>` tag.

```
<apex:page docType="html-5.0" sidebar="false" showHeader="false"
    contentType="text/html" applyHtmlTag="false" applyBodyTag="false"
    standardStylesheets="false" cache="true">
<html>
    <head>
        <meta charset="utf-8"></meta>
        <meta name="viewport"
            content="initial-scale=1, maximum-scale=1, user-scalable=no"></meta>

      <apex:includeScript value="{!URLFOR($Resource.Easy,
                                 'cordova/cordova.js')}"
      <apex:includeScript value="{!URLFOR($Resource.Easy,
                                 'libs/force.js')}"  />
      <script>
(function() {
    /* Do login */
    force.login(
        function() {
            console.log("Auth succeeded");
            showContactsList();
        },
        function(error) {
            console.log("Auth failed: " + error);
```

```
        }
    );

    /* This method will render a list of contacts from current salesforce org */
    var showContactsList = function() {

        fetchRecords(function(data) {
            var contacts = data.records;

            var listItemsHtml = '';
            for (var i=0; i < contacts.length; i++) {
                listItemsHtml += ('<li class="table-view-cell"><
                    div class="media-body">' + contacts[i].Name + '</div></li>');
            }

            document.querySelector('#contacts').innerHTML = listItemsHtml;
        })
    }

    /* This method will fetch a list of contact records from salesforce.
    Just change the soql query to fetch another sobject. */
    var fetchRecords = function (successHandler) {
        var soql = 'SELECT Id, Name FROM Contact LIMIT 10';
        force.query(soql, successHandler, function(error) {
            alert('Failed to fetch contacts: ' + error);
        });
    };

})();
      </script>
    </head>
    <body>

      <header>
         <h1>Hello, Visualforce!</h1>
      </header>

        <!-- Placeholder to add Contacts list -->

      <ul id="contacts">
      </ul>

      <p>Welcome to Mobile SDK.</p>
    </body>
</html>

</apex:page>
```

Note:

- Using the REST API—even from a Visualforce page—consumes API calls.
- Salesforce API calls made through a Mobile SDK container or through a Cordova webview do not require proxy services. Cordova webviews disable same-origin policy, so you can make API calls directly. This exemption applies to all Mobile SDK hybrid and native apps.

Additional Options

You can use the Mobile Sync in HTML5 apps. Just include the required JavaScript libraries as static resources. Take advantage of the model and routing features. Offline access is disabled for this use case. See Using Mobile Sync to Access Salesforce Objects.

Salesforce Developer Marketing provides developer mobile packs that can help you get a quick start with HTML5 apps.

Offline Limitations

Read these articles for tips on using HTML5 with Salesforce Platform offline.

- https://developer.salesforce.com/blogs/developer-relations/2011/06/using-html5-offline-with-forcecom.html
- https://developer.salesforce.com/blogs/developer-relations/2013/03/using-javascript-with-force-com.html

Hybrid Apps Quick Start

Hybrid apps give you the ease of JavaScript and HTML5 development while leveraging Salesforce Mobile SDK

Important: In 2020, the App Store has removed `UIWebView` architecture from all app submissions in favor of `WKWebView`. As a result, Mobile SDK hybrid apps for iOS can run only on Mobile SDK 8.1 or later. See Removing UIWebView from iOS Hybrid Apps.

If you're comfortable with the concept of hybrid app development, use the following steps to get going quickly.

1. To develop Android hybrid apps for Mobile SDK 11.1, you need:
   - Cordova 12.0.1.
   - Cordova CLI 12.0.0 or later.
   - Java JDK 11.0.11+9 or later—www.oracle.com/downloads.
   - Latest version of Android Studio —developer.android.com/sdk.
   - Android SDK, including all APIs in the following range:
     - Minimum API: Android Nougat (API 24)
     - Target API: Android 13 (API 33)
   - Android SDK Tools
   - Android Virtual Device (AVD)
2. To develop iOS hybrid apps for Mobile SDK 11.1, you need:
   - Cordova 7.0.1.
   - Cordova CLI 12.0.0 or later.
   - Xcode version: 14 or later. (We recommend the latest version.)
   - iOS SDK:
     - Deployment target: iOS 15
     - Base SDK: iOS 16
   - CocoaPods (any version from 1.8 to no declared maximum—see cocoapods.org).
3. Install Mobile SDK.
   - Android Preparation

- iOS Preparation

4. If you don't already have a connected app, see Creating a Connected App. For OAuth scopes, select `api`, `web`, and `refresh_token`.

   Note: When specifying the Callback URL, there's no need to use a real address. Use any value that looks like a URL, such as `myapp:///mobilesdk/oauth/done`.

5. Create a hybrid app.
   - Follow the steps at Create Hybrid Apps. Use *`hybrid_local`* for the application type.

6. Run your new app.
   - Build and Run Your Hybrid App on Android
   - Build and Run Your Hybrid App On iOS

   .

Creating Hybrid Apps

Hybrid apps combine the ease of HTML5 Web app development with the power and features of the native platform. They run within a Salesforce mobile container—a native layer that translates the app into device-specific code—and define their functionality in HTML5 and JavaScript files. These apps fall into one of two categories:

- **Hybrid local**—Hybrid apps developed with the `force.js` library wrap a Web app inside the mobile container. These apps store their HTML, JavaScript, and CSS files on the device.
- **Hybrid remote** — Hybrid apps developed with Visualforce technology deliver Apex pages through the mobile container. These apps store some or all of their HTML, JavaScript, and CSS files either on the Salesforce server or on the device (at `http://localhost`).

In addition to providing HTML and JavaScript code, you also must maintain a minimal container app for your target platform. These apps are little more than native templates that you configure as necessary.

If you're creating libraries or sample apps for use by other developers, we recommend posting your public modules in a version-controlled online repository such as GitHub (https://github.com). For smaller examples such as snippets, GitHub provides *gist*, a low-overhead code sharing forum (https://gist.github.com).

SEE ALSO:

Updating Mobile SDK Apps (5.0 and Later)

About Hybrid Development

JavaScript development in a browser is straightforward. After you've altered the code, you merely refresh the browser to see your changes. Developing hybrid apps with the Mobile SDK container requires you to recompile and rebuild after you make changes. For this reason, we recommend you develop your hybrid app directly in a browser, and only run your code in the container in the final stages of testing.

We recommend developing in a browser such as Google Chrome that comes bundled with developer tools. These tools let you access the symbols and code of your web application during runtime.

Mobile SDK JavaScript libraries give you a choice to code with traditional callback functions:

```
traditionalCallbackMethod(args, onSuccess, onFailure)
```

or using promises:

```
promiseBasedMethod(args).then(onSuccess).catch(onFailure)
```

Using the callback function, you can write:

```
self.smartstoreClient.removeSoup(soupName,
    onSuccessRemoveSoup(soupName),
    onErrorRemoveSoup(soupName));
...

function onSuccessRemoveSoup(name) {…}
function onErrorRemoveSoup(name) {…}
```

Promises help you keep your asynchonous code inline, making it easier to follow. Using the promise function, you can rewrite the callback code like this:

```
self.smartstoreClient.removeSoup(soupName)
    .then(function(soupName) {
        ...
    })
    .catch(function(soupName) {
   ...
    })
```

Building Hybrid Apps With Cordova

Salesforce Mobile SDK 11.1 provides a hybrid container that uses a specific version of Apache Cordova for each platform (7.0.1 for iOS, 12.0.1 for Android). Architecturally, Mobile SDK hybrid apps are Cordova apps that use Salesforce Mobile SDK as a Cordova plug-in. Cordova provides a simple command line tool for updating the plug-in in an app. To read more about Cordova benefits, see https://cordova.apache.org/.

Using Forcehybrid

For creating hybrid apps, Mobile SDK provides the forcehybrid npm utility. This utility works with the Cordova command line to build hybrid Mobile SDK projects. With forcehybrid, you can create hybrid projects for iOS, Android, or both in a single pass.

Forcehybrid gives you two ways to create your app.

- Specify the type of application you want, along with basic configuration data.

  OR

- Use an existing Mobile SDK app as a template. You still provide the basic configuration data.

You can use forcehybrid interactively at the command line, or in script mode with command line parameters. To see usage information, type *`forcehybrid`* without arguments.

Using `forcehybrid create` Interactively

To enter application options interactively at a command prompt, type *`forcehybrid create`*. The forcehybrid utility then prompts you for each configuration option. For example:

```
$ forcehybrid create
Enter the target platform(s) separated by commas (ios, android): ios,android
```

```
Enter your application type (hybrid_local, hybrid_remote): <press RETURN>
Enter your application name: LocalHybridTest
Enter your package name: com.myhybrid.ios
Enter your organization name (Acme, Inc.): BestApps.com
Enter output directory for your app (leave empty for the current directory): LocalHybridTest
```

This example creates a hybrid local app named "LocalHybridTest" in the `./LocalHybridTest/` directory, with iOS and Android targets.

Note: Although `forcehybrid create` sets up hybrid projects for the platforms you specify, the app isn't ready for building until you've finished the setup at the Cordova command line. See Create Hybrid Apps

Using `forcehybrid create` in Script Mode

In script mode, you enter your parameters in a single command line instruction:

```
$ forcehybrid create --platform=ios,android --apptype=hybrid_local --appname=packagetest
--packagename=com.test.my_new_app --organization="Acme Widgets, Inc."
--outputdir=PackageTest
```

This example creates a hybrid local app named "packagetest" in the `./PackageTest/` directory, with iOS and Android targets. Here's a description of the available options.

```
Usage:

# create ios/android hybrid_local or hybrid_remote mobile application
forcehybrid create
    --platform=Comma separated platforms (ios, android)
    --apptype=Application Type (hybrid_local, hybrid_remote)
    --appname=Application Name
    --packagename=App Package Identifier (e.g. com.mycompany.myapp)
    --organization=Organization Name (Your company's/organization's name)
    [--startpage=App Start Page (The start page of your remote app. Only required for
hybrid_remote)]
    [--outputdir=Output Directory (Leave empty for current directory)]
```

Using `forcehybrid createWithTemplate`

The *`forcehybrid createWithTemplate`* command is identical to *`forcehybrid create`* except that it asks for a GitHub repo URI instead of an app type. You set this path to point to any repo directory that contains a Mobile SDK app that can be used as a template. Your template app can be any supported Mobile SDK app type. The force script changes the template's identifiers and configuration to match the values you provide for the other parameters.

Before you use `createWithTemplate`, it's helpful to know which templates are available. To find out, type *`forcehybrid listtemplates`*. This command prints a list of templates provided by Mobile SDK. Each listing includes a brief description of the template and its GitHub URI. For example:

```
Available templates:

1) Basic hybrid local applicationforcehybrid createwithtemplate
    --templaterepouri=HybridLocalTemplate
2) Basic hybrid remote applicationforcehybrid createwithtemplate
    --templaterepouri=HybridRemoteTemplate
```

Once you've found a template's URI, you can plug it into the forcehybrid command line. Here's command line usage information for `forcehybrid createWithTemplate`:

```
# create ios/android hybrid_local or hybrid_remote mobile application from a template
forcehybrid createWithTemplate
    --platform=Comma separated platforms (ios, android)
    --templaterepouri=Template repo URI
    --appname=Application Name
    --packagename=App Package Identifier (e.g. com.mycompany.myapp)
    --organization=Organization Name (Your company's/organization's name)
    [--outputdir=Output Directory (Leave empty for current directory)]
```

For any template in the `SalesforceMobileSDK-Templates` repo, you can drop the path for `templaterepouri`—just the template name will do. For example, consider the following command line call:

```
forcehybrid createWithTemplate
--platform=android
--templaterepouri=HybridLocalTemplate
--appname=MyHybrid
--packagename=com.mycompany.hybridlocal
--organization="Acme Software, Inc."
```

This call replicates the hybrid local template app. It recreates the app in the current directory with the same source code and resources as the template app. Forcehybrid changes the app name to "MyHybrid" throughout the project. (This `createwithtemplate` call is equivalent to creating a `hybrid_local` app with forcehybrid.)

How the Forcehybrid Script Generates New Apps

The forcehybrid script

- Generates apps with the Cordova command line.
- Downloads the template app and a `bootconfig.json` file from GitHub.
- Downloads the SalesforceMobileSDK Cordova plugin from GitHub. This plugin delivers the Mobile SDK libraries as Android and iOS library projects.

Create Hybrid Apps

Once you've installed forcehybrid and the Cordova command line, you're ready to create functioning hybrid apps.

Set Up Your Tools

If you haven't already set up the required tools, use the following instructions. Or, if you prefer, complete the Set Up Your Mobile SDK Developer Tools Trailhead project and win Trailhead points for your efforts. Afterwards, return here and pick up at Create a Hybrid Mobile SDK App.

Note: Some of the following steps use the `sudo` keyword. This keyword is required in Mac OS X if you lack read/write permissions. Omit the `sudo` command if you're sure you don't need it or if you're installing on Windows.

1. Make sure that you meet the requirements listed at Development Prerequisites for iOS and Android on page 22. Hybrid development requires at least one native environment, which can be either iOS or Android.
2. Open a command prompt or terminal window.
3. Run *`cordova -v`*.

4. If *cordova -v* reports that the command is not found, install the Cordova command line, version 12.0.0 or later:

```
sudo npm install -g cordova
```

5. Install the forcehybrid npm package:

```
sudo npm install -g forcehybrid
```

If you previously installed an earlier version of forcehybrid, be sure to uninstall it before reinstalling forcehybrid.

Create a Hybrid Mobile SDK App

1. At a command prompt or terminal window, run *forcehybrid create*.
   - For platform, enter "ios", "android", or "ios,android".
   - For application type:
     - Specify `hybrid_local` for a Cordova hybrid app that stores its code on the mobile device.
     - Specify `hybrid_remote` for a Cordova hybrid app that runs a Visualforce app on the server.
   - Provide your own app name, package name, and organization strings.
   - (Hybrid remote apps only) For start page, specify the relative URL of your Apex landing page—for example, `apex/BasicVFPage`.
2. If you're importing HTML, JavaScript, CSS, or `bootconfig.json` files, put them in your project's *<outputdir>*`/www/` directory.

   Important: Do not include `cordova.js`, `cordova.force.js`, or any Cordova plug-ins.

3. In your project directory, open the `www/bootconfig.json` file in a UTF-8 compliant text editor and replace the values of the following properties:
   - `remoteAccessConsumerKey`—Replace the default value with the consumer key from your connected app
   - `oauthRedirectURI`—Replace the default value with the callback URL from your connected app
4. *cd* to your app's project directory. The force script prints the directory name to the screen when it has finished creating your project. For example: "Your application project is ready in *<project directory name>*."
5. For each additional Cordova plug-in you want to add, type:

```
cordova plugin add <plug-in repo or plug-in name>
```

   Note: Go to https://plugins.cordova.io to search for available plug-ins.

6. (Optional—Mac only) To add a second platform "after the fact":
   - To add iOS support, type:

```
cordova platform add ios@7.0.1
```

   - To add Android support, type:

```
cordova platform add android@12.0.1
```

7. Type:

```
cordova prepare
```

   to deploy your web assets to their respective platform-specific directories under the `www/` directory.

   Important: During development, always run `cordova prepare` after you've changed the contents of the `www/` directory, to deploy your changes to the platform-specific project folders.

See "The Command-Line Interface" in the Cordova 3.5 documentation for more information on the Cordova command line.

Build and Run Your Hybrid App on Android

Before building, be sure that you've installed Android Studio, including Android SDK and at least one Android emulator. Refer to the Android requirements for Mobile SDK to make sure you install the correct versions of the Android components.

After you've run `cordova prepare`, build and run the project.

To run the app in Android Studio:

1. Launch Android Studio.
2. From the welcome screen, select **Import project (Eclipse ADT, Gradle, etc.)**. Or, if Android Studio is already running, select **File** > **New** > **Import Project**.
3. Select `<your_project_dir>/platforms/android` and click **OK**. If you're prompted to use the Gradle wrapper, accept the prompt.
4. After the build finishes, select the `android` target and click **Run 'android'** from either the menu or the toolbar.
5. Select a connected Android device or emulator.

Important: If Android Studio offers to update your Gradle wrapper version, accept the offer. After the process finishes, Android Studio automatically re-imports your project.

Build and Run Your Hybrid App On iOS

After you've run `cordova prepare` on an iOS hybrid app, you can open the project in Xcode to run the app in an iOS simulator.

To run the app in Xcode:

1. In Xcode, select **File** > **Open**.
2. Navigate to the `platforms/ios/` directory in your new app's directory.
3. Double-click the `<app name>.xcodeworkspace` file.
4. Click the Run button in the upper left corner, or press *COMMAND-R*.

Developing Hybrid Remote Apps

You can easily convert the FileExplorer SDK sample (github.com/forcedotcom/SalesforceMobileSDK-Shared/tree/master/samples/fileexplorer), which is a hybrid local app, into a hybrid remote app. To convert the app, you redefine the main HTML page as a Visualforce page that is delivered from the server. You can then bundle the CSS and JavaScript resources with the app so that they're stored on the device.

Let's start by creating the Visualforce page.

1. In your Salesforce Developer Edition org, create a Visualforce page named "FileExplorer" with the following attributes.

```
<apex:page docType="html-5.0" showHeader="false" sidebar="false">
<!-- Paste content of FileExplorer.html here, but remove the "<!DOCTYPE html>" directive
 -->
</apex:page>
```

2. Copy the contents of the `samples/fileexplorer/FileExplorer.html` file into the FileExplorer Visualforce page.
3. Delete the `<!DOCTYPE html>` directive at the top of the inserted content.
4. Save your work.

Next, create a hybrid remote app to contain the sample code.

1. *`cd`* to the directory where you want to develop your app. The only requirement is that this directory cannot already contain a subdirectory named "fileexplorer".
2. In a Terminal window or command prompt, run *`forcehybrid create`* with the following values:

Platform:	*`ios,android`*
Application type:	*`hybrid_remote`*
Application name:	*`fileexplorer`*
Package name:	*`com.salesforce.fileexplorer`*
Organization name:	*`Acme Apps, Inc.`*
Start page:	*`apex/FileExplorer`*
Output directory:	*`<press RETURN>`*

3. In a text editor, open `fileexplorer/www/bootconfig.json` and change the following properties as follows:

```
"isLocal": false,
"startPage": "apex/FileExplorer",
```

   These settings configure your app to be a hybrid remote app.
4. Return to your Terminal window or command prompt, and then type:

```
cordova prepare
```

Done! To run the Android target, import the *`<my_app_directory>`*`/fileexplorer/platforms/android` folder into Android Studio and run the app. Or, to run the iOS target, import the *`<my_app_directory>`*`/fileexplorer/platforms/ios/fileexplorer.xcworkspace` file into Xcode and run the app. When you test this sample, be sure to log in to the organization where you created the Visualforce page.

Using `localhost` in Hybrid Remote Apps for iOS

Beginning with version 5.0, Mobile SDK followed Apple's mandate and deprecated the `UIWebView` class in favor of `WKWebView`.

To comply with the App Store's upcoming policy of rejecting apps that use `UIWebView`, Mobile SDK 8.1 removes all references to that class. As a result, you can no longer use `localhost` in Mobile SDK hybrid remote apps.

Hybrid Sample Apps

Salesforce Mobile SDK provides hybrid samples that demonstrate how to use Mobile SDK features in JavaScript. We provide hybrid samples two ways:

- As platform-specific apps with native wrappers. We provide these wrappers for a limited subset of our hybrid samples. You can access the iOS samples through the Mobile SDK workspace (`SalesforceMobileSDK.xcodeproj`) in the root directory of the SalesforceMobileSDK-iOS GitHub repository. Also, you can access the Android samples from the `hybrid/SampleApps` directory of the SalesforceMobileSDK-Android repository.
- As platform-agnostic web apps including only the HTML5, JavaScript, CSS source code. These apps include all of our hybrid samples and provide the basis for the platform-specific hybrid apps. You can download these sample apps from the SalesforceMobileSDK-Shared GitHub repo and build them using the Cordova command line.

Android Hybrid Sample Wrappers

- **AccountEditor**: Demonstrates how to synchronize offline data using the `mobilesync.js` library.
- **MobileSyncExplorerHybrid**: Demonstrates how to synchronize offline data using the Mobile Sync plugin.
- **NoteSync**: Demonstrates how to use non-REST APIs to retrieve Salesforce Notes.

iOS Hybrid Sample Wrappers

- **AccountEditor**: Demonstrates how to synchronize offline data using the `mobilesync.js` library.
- **MobileSyncExplorer**: Demonstrates how to synchronize offline data using the Mobile Sync plugin.
- **NoteSync**: Demonstrates how to use non-REST APIs to retrieve Salesforce Notes.

Source-only Hybrid Sample Apps

Salesforce Mobile SDK provides the following platform-agnostic hybrid sample apps in the the SalesforceMobileSDK-Shared GitHub repository.

- **accounteditor**: Uses the Mobile Sync to access Salesforce data.
- **contactexplorer**: Uses Cordova to retrieve local device contacts. It also uses the `force.js` toolkit to implement REST transactions with the Salesforce REST API. The app uses the OAuth2 support in Salesforce SDK to obtain OAuth credentials and then propagates those credentials to `force.js` by sending a javascript event.
- **fileexplorer**: Demonstrates the Files API.
- **mobilesyncexplorer**: Demonstrates using `mobilesync.js`, rather than the Mobile Sync plug-in, for offline synchronization.
- **notesync**: Uses non-REST APIs to retrieve Salesforce Notes.
- **simplesyncreact:**: Demonstrates a React Native app that uses the Mobile Sync plug-in.
- **smartstoreexplorer**: Lets you explore SmartStore APIs.
- **userandgroupsearch**: Lets you search for users in groups.
- **userlist**: Lists users in an organization. This is the simplest hybrid sample app.
- **usersearch**: Lets you search for users in an organization.
- **vfconnector**: Wraps a Visualforce page in a native container. This example assumes that your org has a Visualforce page called `BasicVFTest`. The app first obtains OAuth login credentials using the Salesforce SDK OAuth2 support and then uses those credentials to set appropriate webview cookies for accessing Visualforce pages.

Build Hybrid Sample Apps

To build hybrid apps from the `samples` directory of the SalesforceMobileSDK-Shared repository, you use forcehybrid and the Cordova command line. You create a `hybrid_local` or `hybrid_remote` app and then add the web assets—HTML, JavaScript, and CSS files—and the `bootconfig.json` file from the Shared repo.

Note: The ContactExplorer sample requires the `cordova-plugin-contacts` and `cordova-plugin-statusbar` plug-ins.

The other hybrid sample apps do not require special Cordova plug-ins.

To build one of the sample apps:

1. Open a command prompt or terminal window.
2. Clone the shared repo:

```
git clone https://github.com/forcedotcom/SalesforceMobileSDK-Shared
```

3. Use forcehybrid to create an app.
   - For platform, enter one or both platform names: “ios”, “android”, or “ios,android”.
   - For application type (or the `apptype` parameter), enter “hybrid_local”.
4. Change to your new app directory:

```
cd <app_target_directory>
```

5. For each additional Cordova plug-in you want to add, type:

```
cordova plugin add <plug-in repo or plug-in name>
```

Note: Go to https://plugins.cordova.io to search for available plug-ins.

6. (Optional—Mac only) To add iOS support to an Android project “after the fact”:

```
cordova platform add ios@7.0.1
```

7. (Optional—Mac only) To add Android support to an iOS project “after the fact”:

```
cordova platform add android@12.0.1
```

8. Copy the sample source files to the `www` folder of your new project directory.

   On Mac:

```
cp -RL <local path to SalesforceMobileSDK-Shared>/SampleApps/<template>/* www/
```

   On Windows:

```
copy <local path to SalesforceMobileSDK-Shared>\SampleApps\<template>\*.* www
```

   If you’re asked, affirm that you want to overwrite existing files.
9. Do the final Cordova preparation:

```
cordova prepare
```

Note:

- Android Studio refers to forcehybrid projects by the platform name ("android"). For example, to run your project, select "android" as the startup project and then click Run.
- On Windows, Android Studio sets the default project encoding to `windows-1252`. This setting conflicts with the `UTF-8` encoding of the forcehybrid Gradle build files. For best results, change the default project encoding to *`UTF-8`*.
- On Windows, be sure to run Android Studio as administrator.

Running the ContactExplorer Hybrid Sample

Let's look at the ContactExplorer sample app, which is included in Mobile SDK. You can do this exercise on Mac OS or Windows, but you can fully validate the iOS target only on a Mac.

Before starting this exercise, be sure that you have:

- A directory to contain the `SalesforceMobileSDK-Shared` cloned repo—your root directory, or any other easily accessible location.
- A directory for creating and developing Mobile SDK hybrid projects. Since Cordova projects can contain both iOS and Android targets, it's a good idea to put them in a platform-neutral directory.

To begin, clone the shared repo, then create an app with forcehybrid.

1. At a command prompt or Terminal window, *`cd`* to the directory where you plan to clone the shared repo.
2. Run the following command.

```
git clone https://github.com/forcedotcom/SalesforceMobileSDK-Shared.git
```

3. *`cd`* to the directory where you plan to develop your hybrid project.
4. Run *`forcehybrid create`* with the following values:

```
Enter the target platform(s) separated by commas (ios, android): ios,android
Enter your application type (hybrid_local or hybrid_remote, leave empty for hybrid_local):
 <press RETURN>
Enter your application name: contactsApp
Enter the package name for your app (com.mycompany.myapp): com.acmeapps.contactexplorer
Enter your organization name (Acme, Inc.): AcmeApps.com
Enter output directory for your app (leave empty for the current directory): <press
RETURN>
```

Now that you have a generic hybrid project, you can add the `contactexplorer` sample code to it.

5. Run the following commands, making sure to replace the placeholder in the *`cp`* command with your local path.

```
cd contactsApp
cordova plugin add cordova-plugin-contacts
cordova plugin add cordova-plugin-statusbar
cp -RL <local path to SalesforceMobileSDK-Shared>/samples/contactexplorer/* www/
cordova prepare
```

Note: *Windows users*: On Windows, substitute the `copy` command for the `cp` Unix command. Be aware, however, that files in the `js` and `css` subfolders of /samples/contactexplorer/ are aliases to source files on other paths. Make sure that you copy the source files themselves rather than their aliases. Here's an example:

```
cd contactsApp
cordova plugin add cordova-plugin-contacts
cordova plugin add cordova-plugin-statusbar
rem Make a path variable
set SHAREDPATH=C:\SalesforceMobileSDK-Shared\
md www
cd www
md css
copy %SHAREDPATH%\samples\common\jquery.mobile-1.3.1.min.css css
md js
copy %SHAREDPATH%\test\MockCordova.js js
copy %SHAREDPATH%\libs\cordova.force.js js
copy %SHAREDPATH%\libs\force.js js
copy %SHAREDPATH%\dependencies\jquery\jquery.min.js js
copy %SHAREDPATH%\samples\common\jquery.mobile-1.3.1.min.js js
cordova prepare
```

The forcedroid script and the ensuing commands create an iOS project and an Android project, both of which wrap the ContactExplorer sample app. Now we're ready to run the app on one of these platforms. If you're using an iOS device, you must configure a profile for the simulator, as described in the Xcode User Guide at developer.apple.com/library. Similarly, Android devices must be set up as described at developer.android.com/tools.

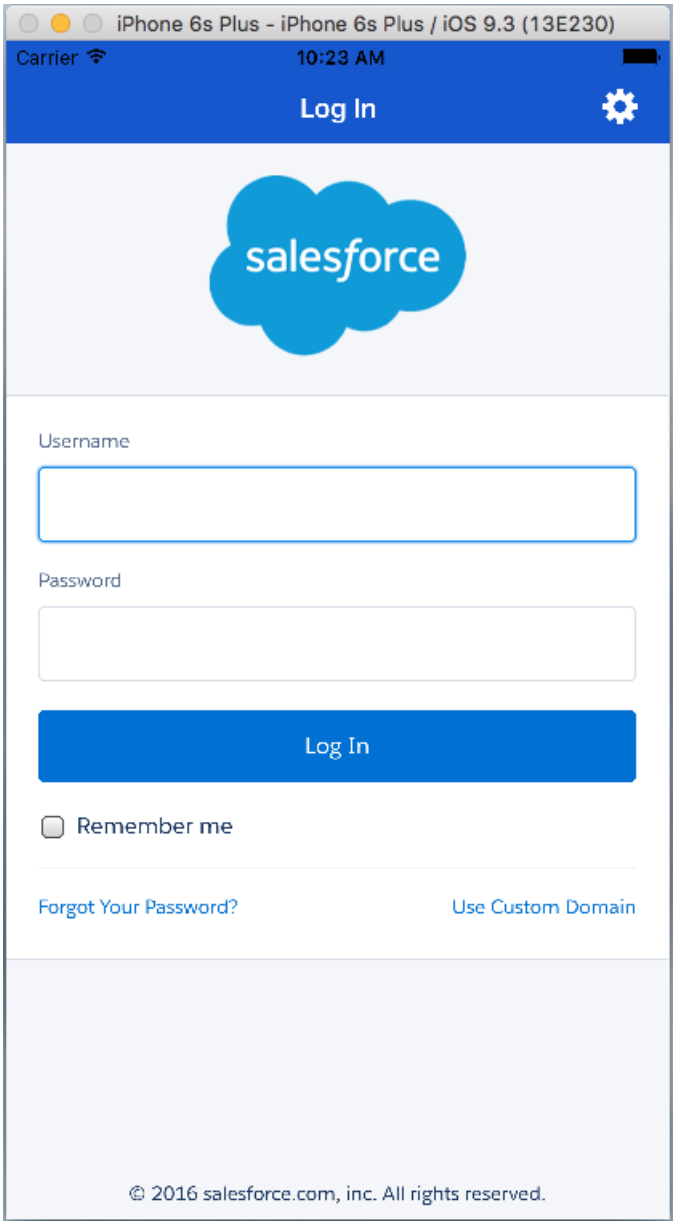
To run the app on iOS:

1. *cd* to `platforms/ios/`.
2. Run the following command: *open contactsApp.xcworkspace*
3. In Xcode, click **Run**.

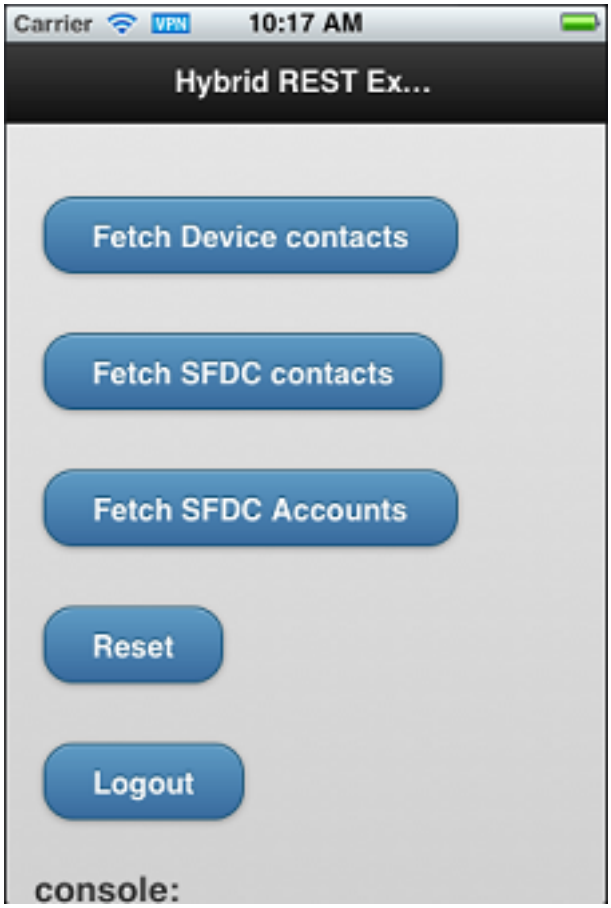
To run the app on Android:

1. In Android Studio, import or open the *<your-hybrid-projects-directory>*`/contactsApp/platforms/android` project.
2. Click **Run**.

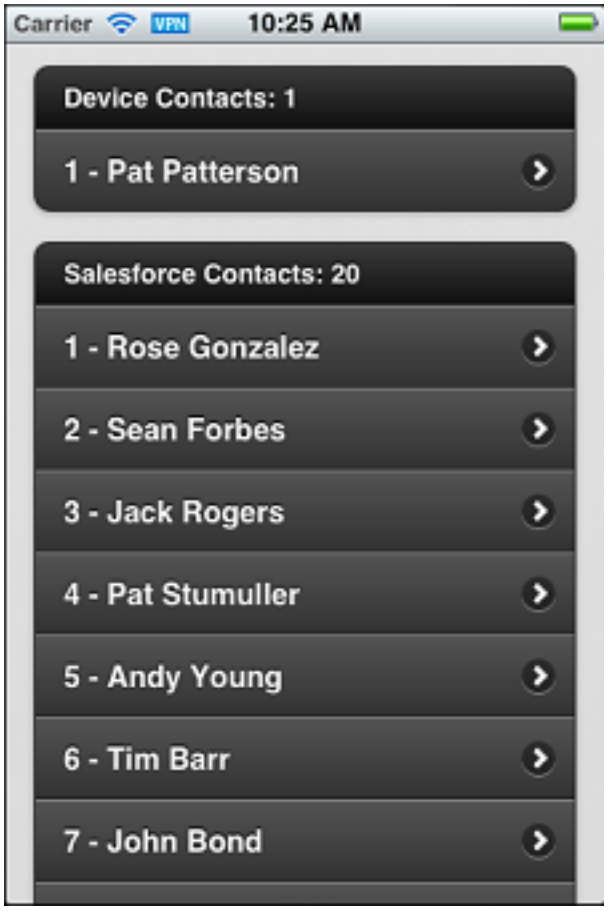
When you run the app, after an initial splash screen, you see the Salesforce login screen.



Log in with your Developer Edition org username and password. To allow the app to access your Salesforce data, tap **Allow**. Now that you're in the app, you can retrieve lists of contacts and accounts. Tap **Fetch SFDC contacts** to retrieve Salesforce contact names or **Fetch SFDC Accounts** to retrieve account names from your DE organization.



With each tap, the app appends rows to an infinite list. Scroll down to see the full list.



Let's take a closer look at how the app works.

To initiate a user session with force.js, you call `force.login()`. After the user logs in to an app running in the container, the network plug-in refreshes tokens as necessary when the app tries to access Salesforce resources. The following code, adapted from the ContactExplorer sample, demonstrates a typical `force.login()` implementation.

When the device notifies that it's ready, you call the `force.login()` method to post the login screen.

```
/* Do login */
force.login(
    function() {
        console.log("Auth succeeded");
        // Call your app's entry point
        // ...
    },
    function(error) {
        console.log("Auth failed: " + error);
    }
);
```

After completing the login process, the sample app displays `index.html` (located in the www folder). When the page has completed loading and the mobile framework is ready, the `jQuery(document).ready()` function calls `regLinkClickHandlers()`.

This function (in `inline.js`) sets up click handlers for the various functions in the sample app. For example, the `#link_fetch_sfdc_contacts` handler runs a query using the `force` object.

```
$j('#link_fetch_sfdc_contacts').click(function() {
    logToConsole("link_fetch_sfdc_contacts clicked");
    force.query("SELECT Name FROM Contact LIMIT 25",
        onSuccessSfdcContacts, onErrorSfdc);
});
```

The `force` object is set up during the initial OAuth 2.0 interaction, and gives access to REST API in the context of the authenticated user. Here, we retrieve the names of all the contacts in the DE organization. `onSuccessSfdcContacts()` then renders the contacts as a list on the `index.html` page.

```
$j('#link_fetch_sfdc_accounts').click(function() {
    logToConsole("link_fetch_sfdc_accounts clicked");
    force.query("SELECT Name FROM Account LIMIT 25",
        onSuccessSfdcAccounts, onErrorSfdc);
});
```

Similarly to the `#link_fetch_sfdc_contacts` handler, the `#link_fetch_sfdc_accounts` handler fetches Account records via REST API. The `#link_reset` and`#link_logout` handlers clear the displayed lists and log out the user, respectively.

Notice that the app can also retrieve contacts from the device—something that an equivalent web app would be unable to do. The following click handler retrieves device contact query by calling the Cordova contacts plug-in.

```
$j('#link_fetch_device_contacts').click(function() {
    logToConsole("link_fetch_device_contacts clicked");
    var options      = new ContactFindOptions();
    // empty search string returns all contacts
    options.filter   = "";
    options.multiple = true;
    options.hasPhoneNumber = true;
    var fields       =
        [navigator.contacts.fieldType.displayName,
            navigator.contacts.fieldType.name];
    navigator.contacts.find(fields, onSuccessDevice,
        onErrorDevice, options);
});
```

This handler uses the `ContactFindOptions` and `navigator.contacts` objects from `cordova-plugin-contacts` to refine and execute a search. It calls `navigator.contacts.find()` to retrieve a list of contacts with phone numbers from the device. The `onSuccessDevice()` function (not shown here) renders the contact list into the `index.html` page.

SEE ALSO:

- Build and Run Your Hybrid App On iOS
- Build and Run Your Hybrid App on Android

Debugging Hybrid Apps On a Mobile Device

You can debug hybrid apps while they're running on a mobile device. How you do it depends on your development platform.

If you run into bugs that show up only when your app runs on a real device, you'll want to use your desktop developer tools to troubleshoot those issues. It's not always obvious to developers how to connect the two runtimes, and it's not well documented in some cases. Here are general platform-specific steps for attaching a Web app debugger on your machine to a running app on a connected device.

Debugging a Hybrid App On an Android Device

To debug hybrid apps on Android devices, use Google Chrome.

The following steps summarize the full instructions posted at https://developer.chrome.com/devtools/docs/remote-debugging

1. Enable USB debugging on your device: https://developer.chrome.com/devtools/docs/remote-debugging
2. Open Chrome on your desktop (development) machine and navigate to: `chrome://inspect`
3. Select **Discover USB Devices**.
4. Select your device.
5. To use your device to debug a web application that's running on your development machine:
   a. Click **Port forwarding…**.
   b. Set the device port and the localhost port.
   c. Select **Enable port forwarding**. See https://developer.chrome.com/devtools/docs/remote-debugging#port-forwarding for details.

Debug a Hybrid App Running on an iOS Device

To debug hybrid apps on real or simulated iOS devices, use Safari on the desktop and the device.

1. Open Safari on the desktop.
2. Select **Safari** > **Preferences**.
3. Click the **Advanced** tab.
4. Click **Show Develop menu in menu bar**.
5. If you're using the iOS simulator:
   - In the system task bar, press CONTROL and click the Xcode icon, then select **Open Developer Tool** > **Simulator**.
   - Or, in a Terminal window, type `open -a Simulator.app`.
6. In the iOS Simulator menu, select **File** > **Open Simulator**.
7. Select a device.
8. Open Safari from the home screen of the device or iOS Simulator.
9. Navigate to the location of your web app.
10. In Safari on your desktop, select **Developer** > ***<your device>***, and then select the URL that you opened in Safari on the device or simulator.

    The Web Inspector window opens and attaches itself to the running Safari instance on your device.

Controlling the Status Bar in iOS 7 Hybrid Apps

In iOS 7 you can choose to show or hide the status bar, and you can control whether it overlays the web view. You use the Cordova status bar plug-in to configure these settings. By default, the status bar is shown and overlays the web view in Salesforce Mobile SDK 2.3 and later.

To hide the status bar, add the following keys to the application plist:

```
<key>UIStatusBarHidden</key>
<true/>
<key>UIViewControllerBasedStatusBarAppearance</key>
<false/>
```

For an example of a hidden status bar, see the AccountEditor sample app.

To control status bar appearance--overlaying, background color, translucency, and so on--add org.apache.cordova.statusbar to your app:

```
cordova plugin add org.apache.cordova.statusbar
```

You control the appearance either from the `config.xml` file or from JavaScript. See https://github.com/apache/cordova-plugin-statusbar for full instructions. For an example of a status bar that doesn't overlay the web view, see the ContactExplorer sample app.

SEE ALSO:

Hybrid Sample Apps

JavaScript Files for Hybrid Apps

External Dependencies

Mobile SDK uses the following external dependencies for various features of hybrid apps.

External JavaScript File	Description
jquery.js	Popular HTML utility library
underscore.js	Mobile Sync support
backbone.js	Mobile Sync support

Which JavaScript Files Do I Include?

Beginning with Mobile SDK 2.3, the Cordova utility copies the Cordova plug-in files your application needs to your project's platform directories. You don't need to add those files to your www/ folder.

Files that you include in your HTML code (with a <script> tag> depend on the type of hybrid project. For each type described here, include all files in the list.

For basic hybrid apps:

- `cordova.js`

To make REST API calls from a basic hybrid app:

- `cordova.js`
- `force.js`

To use Mobile Sync in a hybrid app:

- `jquery.js`
- `underscore.js`
- `backbone.js`
- `cordova.js`
- `force.js`
- `mobilesync.js`

Versioning and JavaScript Library Compatibility

In hybrid applications, client JavaScript code interacts with native code through Cordova (formerly PhoneGap) and SalesforceSDK plug-ins. When you package your JavaScript code with your mobile application, your testing assures that the code works with native code. However, if the JavaScript code comes from the server—for example, when the application is written with VisualForce—harmful conflicts can occur. In such cases you must be careful to use JavaScript libraries from the version of Cordova that matches the Mobile SDK version you're using.

For example, suppose you shipped an application with Mobile SDK 1.2, which uses PhoneGap 1.2. Later, you ship an update that uses Mobile SDK 1.3. The 1.3 version of the Mobile SDK uses Cordova 1.8.1 rather than PhoneGap 1.2. You must make sure that the JavaScript code in your updated application accesses native components only through the Cordova 1.8.1 and Mobile SDK 1.3 versions of the JavaScript libraries. Using mismatched JavaScript libraries can crash your application.

You can't force your customers to upgrade their clients, so how can you prevent crashes? First, identify the version of the client. Then, you can either deny access to the application if the client is outdated (for example, with a "Please update to the latest version" warning), or, preferably, serve compatible JavaScript libraries.

The following table correlates Cordova and PhoneGap versions to Mobile SDK versions.

Mobile SDK version	Cordova or PhoneGap version
1.2	PhoneGap 1.2
1.3	Cordova 1.8.1
1.4	Cordova 2.2
1.5	Cordova 2.3
2.0	Cordova 2.3
2.1	Cordova 2.3
2.2	Cordova 2.3
2.3	Cordova 3.5
3.0	Cordova 3.6
3.1	Cordova 3.6

Mobile SDK version	Cordova or PhoneGap version
3.2	Cordova 3.6
3.3	Cordova 3.6
4.0	Cordova 5.0.0 (for Android), 3.9.2 (for iOS)
4.1	Cordova 5.0.0 (for Android), 3.9.2 (for iOS)
4.2	Cordova 5.0.0 (for Android), 3.9.2 (for iOS)
4.3	Cordova 5.0.0 (for Android), 4.2.0 (for iOS)

Finding the Mobile SDK Version with the User Agent

You can leverage the user agent string to look up the Mobile SDK version. The user agent starts with `SalesforceMobileSDK/<version>`. Once you obtain the user agent, you can parse the returned string to find the Mobile SDK version.

You can obtain the user agent on the server with the following Apex code:

```
userAgent = ApexPages.currentPage().getHeaders().get('User-Agent');
```

On the client, you can do the same in JavaScript using the `navigator` object:

```
userAgent = navigator.userAgent;
```

Detecting the Mobile SDK Version with the sdkinfo Plugin

In JavaScript, you can also retrieve the Mobile SDK version and other information by using the `sdkinfo` plug-in. This plug-in, which is defined in the `cordova.force.js` file, offers one method:

```
getInfo(callback)
```

This method returns an associative array that provides the following information:

Member name	Description
`sdkVersion`	Version of the Salesforce Mobile SDK used to build to the container. For example: "1.4".
`appName`	Name of the hybrid application.
`appVersion`	Version of the hybrid application.
`forcePluginsAvailable`	Array containing the names of Salesforce plug-ins installed in the container. For example: "com.salesforce.oauth", "com.salesforce.smartstore", and so on.

The following code retrieves the information stored in the `sdkinfo` plug-in and displays it in alert boxes.

```
var sdkinfo = cordova.require("com.salesforce.plugin.sdkinfo");
sdkinfo.getInfo(new function(info) {
```

```
    alert("sdkVersion->" + info.sdkVersion);
    alert("appName->" + info.appName);
    alert("appVersion->" + info.appVersion);
    alert("forcePluginsAvailable->" +
        JSON.stringify(info.forcePluginsAvailable));
});
```

SEE ALSO:

Example: Serving the Appropriate Javascript Libraries

Example: Serving the Appropriate Javascript Libraries

To provide the correct version of Javascript libraries, create a separate bundle for each Salesforce Mobile SDK version you use. Then, provide Apex code on the server that downloads the required version.

1. For each Salesforce Mobile SDK version that your application supports, do the following.
   a. Create a ZIP file containing the Javascript libraries from the intended SDK version.
   b. Upload the ZIP file to your org as a static resource.

   For example, if you ship a client that uses Salesforce Mobile SDK v. 1.3, add these files to your ZIP file:

   - `cordova.force.js`
   - `SalesforceOAuthPlugin.js`
   - `bootconfig.js`
   - `cordova-1.8.1.js`, which you should rename as `cordova.js`

   Note: In your bundle, it's permissible to rename the Cordova Javascript library as `cordova.js` (or `PhoneGap.js` if you're packaging a version that uses a `PhoneGap-x.x.js` library.)

2. Create an Apex controller that determines which bundle to use. In your controller code, parse the user agent string to find which version the client is using.
   a. In your org, from Setup, click **Develop** > **Apex Class**.
   b. Create a new Apex controller named `SDKLibController` with the following definition.

```
public class SDKLibController {
  public String getSDKLib() {
    String userAgent =
      ApexPages.currentPage().
        getHeaders().get('User-Agent');

    if (userAgent.contains('SalesforceMobileSDK/1.3')) {
      return 'sdklib13';
    }
    // Add if statements for other SalesforceSDK versions
    // for which you provide library bundles.
  }
}
```

3. Create a Visualforce page for each library in the bundle, and use that page to redirect the client to that library.
   For example, for the SalesforceOAuthPlugin library:

a. In your org, from Setup, enter *Visualforce Pages* in the Quick Find box, then select **Visualforce Pages**.

b. Create a new page called "SalesforceOAuthPlugin" with the following definition.

```
<apex:page controller="SDKLibController"
  action="{!URLFor($Resource[SDKLib],
 'SalesforceOAuthPlugin.js')}">
</apex:page>
```

c. Reference the VisualForce page in a <script> tag in your HTML code. Be sure to point to the page you created in step 3b. For example:

```
<script type="text/javascript"
  src="/apex/SalesforceOAuthPlugin" />
```

Note: Provide a separate <script> tag for each library in your bundle.

Managing Sessions in Hybrid Apps

To help resolve common issues that often affect mobile apps, Mobile SDK wraps hybrid apps in native containers. These containers provide seamless authentication and session management by internally managing OAuth token exchanges. However, as popular mobile app architectures evolve, this "one size fits all" approach proves to be too limiting in some cases. For example, if a mobile app uses JavaScript remoting in Visualforce, Salesforce cookies can be lost if the user lets the session expire. These cookies can be retrieved only when the user manually logs back in.

Modern versions of Mobile SDK use reactive session management. "Reactive" means that apps can participate in session management, rather than letting the container do all the work. Apps created before Mobile SDK 1.4, however, used proactive, or container controlled, session management. In the proactive scenario, some types of apps would restart when the session expired, resulting in a less than satisfactory user experience. In the reactive scenario, your app can prevent such restarts by refreshing the session token without interrupting the runtime flow.

If you're upgrading an app from version 1.3 to any later version, you're required to switch to reactive management. To switch to reactive management, adjust your session management settings according to your app's architecture. This table summarizes the behaviors and recommended approaches for common architectures.

App Architecture	Reactive Behavior in Mobile SDK 5.0 and Later	Steps for Upgrading Code
REST API	Refresh from JavaScript using the com.salesforce.plugin.network plug-in	No coding is required for apps that use force.js, which handles network calls natively through the com.salesforce.plugin.network plug-in. Apps that use other frameworks should also adopt the com.salesforce.plugin.network plug-in for network calls.
JavaScript Remoting in Visualforce	Refresh session and CSRF token from JavaScript	Catch the session timeout event, and then either reload the page or load a new iFrame.

Note: In Mobile SDK 5.0 and later, JQuery Mobile, which some hybrid apps use for networking, is no longer supported as a networking option.

The following sections provide code examples for supported architectures.

REST APIs (Including Apex2REST)

Hybrid apps that use REST APIs are required to refresh expired access tokens before each REST call. You can meet this requirement simply by using force.js, which refreshes sessions implicitly through the `com.salesforce.plugin.network` plug-in. With force.js, your app doesn't have to add refresh code.

To initiate a user session with force.js, you call `force.login()`. After the user logs in to an app running in the container, the network plug-in refreshes tokens as necessary when the app tries to access Salesforce resources. The following code, adapted from the ContactExplorer sample, demonstrates a typical `force.login()` implementation.

- When the device notifies that it's ready, call the `force.login()` method to post the login screen.

```
/* Do login */
force.login(
    function() {
        console.log("Auth succeeded");
        // Call your app's entry point
        // ...
    },
    function(error) {
        console.log("Auth failed: " + error);
    }
);
```

Get the complete ContactExplorer sample application here:
https://github.com/forcedotcom/SalesforceMobileSDK-Shared/tree/master/samples/contactexplorer

JavaScript Remoting in Visualforce

For mobile apps that use JavaScript remoting to access Visualforce pages, incorporate the session refresh code into the method parameter list. In JavaScript, use the Visualforce remote call to check the session state and adjust accordingly.

```
<Controller>.<Method>(
    <params>,
    function(result, event) {
        if (hasSessionExpired(event)) {
            // Reload will try to redirect to login page
            // Container will intercept the redirect and
            // refresh the session before reloading the
            // origin page
            window.location.reload();
        } else {
            // Everything is OK.
            // Go ahead and use the result.
            // ...
        },
    {escape: true}
);
```

This example defines `hasSessionExpired()` as:

```
function hasSessionExpired(event) {
    return (event.type == "exception" &&
        event.message.indexOf("Logged in?") != -1);
}
```

Advanced use case: Reloading the entire page doesn't always provide the best user experience. To avoid reloading the entire page, you'll need to:

1. Refresh the access token
2. Refresh the Visualforce domain cookies
3. Finally, refresh the CSRF token

Instead of fully reloading the page as follows:

```
window.location.reload();
```

Do something like this:

```
// Refresh oauth token
cordova.require("com.salesforce.plugin.oauth").authenticate(
    function(creds) {
        // Reload hidden iframe that points to a blank page to
        // to refresh Visualforce domain cookies
        var iframe = document.getElementById("blankIframeId");
        iframe.src = src;

        // Refresh CSRF cookie
        // Get the provider array
        var providers = Visualforce.remoting.Manager.providers;
        // Get the last provider in the arrays (usually only one)
        var provider = Visualforce.remoting.last;
        provider.refresh(function() {
            //Retry call for a seamless user experience
        });

    },
    function(error) {
        console.log("Refresh failed");
    }
);
```

Defer Login

Mobile SDK hybrid apps always present a Salesforce login screen at startup. Sometimes, however, these apps can benefit from deferring authentication until some later point.With a little configuration, you can defer login to any logical place in your app.

Deferred login implementation with `force.js` is a two-step process:

1. Configure the project to skip authentication at startup.
2. In your JavaScript code, call the `force.init()` function, followed by the `force.login()` function, at the point where you plan to initiate authentication.

Step 1: Configure the Project to Skip Authentication

1. In your platform-specific project, open the `www/bootconfig.json` file.
2. Set the `shouldAuthenticate` property to "false".

Step 2: Initiate Authentication in JavaScript

To initiate the authentication process, call the `force.js login()` functions at the point of deferred login. The `force.init()` method is usually necessary only for testing or other non-production scenarios.

```
/* Do login */
force.login(
    function() {
        console.log("Auth succeeded");
        // Call your app's entry point
        // ...
    },
    function(error) {
        console.log("Auth failed: " + error);
    }
);
```

The `force.login()` function takes two arguments: a success callback function and a failure callback function. If authentication fails, your failure callback is invoked. If authentication succeeds, the `force` object caches the access token in its `oauth.access_token` configuration property and invokes your success callback.

CHAPTER 9 Using Lightning Web Components in Hybrid Apps

In Salesforce orgs, developers and admins use Lightning web components to build sophisticated web controls that work well with other Salesforce controls. Mobile SDK has long supported apps that run Visualforce pages in web views. However, until recently, technical hurdles made it difficult to run LWCs in those apps. In version 8.2, Mobile SDK introduces a solution to those hurdles: Developers can now host Lightning web components in custom hybrid remote applications.

Support for LWCs centers around a new template app and a new app type. The LWC template includes a Salesforce DX server that launches the components. You can find the `HybridLwcTemplate` template app at the SalesforceMobileSDK-Templates GitHub repo.

To create an LWC-enabled app, use the Mobile SDK forcehybrid tool specifying the `hybrid_lwc` app type.

```
forcehybrid create
Enter the target platform(s) separated by commas (ios, android):
ios,android
Enter your application type (hybrid_local or hybrid_remote or
hybrid_lwc,
    leave empty for hybrid_local): hybrid_lwc
Enter your application name: helloLwc
Enter your package name: com.acme.apps
Enter output directory for your app (leave empty for the current
directory):
    helloLwc
```

Your generated project includes a `server` folder that contains a Salesforce DX project. Before logging into your hybrid LWC app, deploy this project to your target org. For example, to deploy to a scratch org:

1. If you specified an output directory for your project, *cd* to that directory.

   ```
   cd helloLwc
   ```

2. Create a scratch org.

   ```
   sf org create scratch --definition-file
   server/config/project-scratch-def.json --alias MyOrg
   ```

3. *cd* to your project's `server` directory.

   ```
   cd server
   ```

4. Deploy the project's source files to your scratch org.

   ```
   sf project deploy start --target-org MyOrg
   ```

5. Create a password for your scratch org.

   ```
   sf org generate password --target-org MyOrg
   ```

6. Log in to your app on a virtual or physical device using the user name and password for the new scratch org.

In addition to the Mobile SDK native container app, a hybrid LWC app includes:

- The root Lightning Web Component on a Visualforce page.
- Javascript code for Cordova and Mobile SDK's Cordova plug-ins.
- An Apex class that determines whether to serve iOS or Android libraries.

Interacting with Native Mobile SDK Features

Hybrid LWC apps can interact with the native container through the Mobile SDK Cordova plug-ins. Mobile SDK provides a `mobilesdk` object that you can pass as a property to any Lightning Web Component in your project. For example, to use `mobilesdk` in an LWC Javascript file, add the following annotation:

```
@api mobilesdk;
```

With the `mobilesdk` object, you can call Mobile SDK plug-in functionality. For example, you can make network calls, store data in SmartStore, and use MobileSync. To demonstrate, the template app uses `mobilesdk` in its ContactsList component to make a SOQL query. Here's the pertinent code, from `ContactsList.js`.

```
@api mobilesdk;

connectedCallback() {
    this.loadContacts();
}

loadContacts() {
    let soql = 'SELECT Id, Name, MobilePhone, Department FROM Contact
 LIMIT 100';
    this.mobilesdk.force.query(soql,
        (result) => {
            this.contacts = result.records;
        },
        (error) => {
            this.error = error;
        }
    );
}
```

Other Considerations

- You do most, if not all, of your development in the `server/force-app/main/default/lwc/` directory.
- Developing a LWC for your mobile app is nearly identical to developing for the desktop browser. Differences occur only if you use the `mobilesdk` object to access the native container.
- To change the root component, update `server/force-app/main/default/pages/<app-name>.page` accordingly.

- Hybrid LWC apps use Lightning Out to run Lightning web components in a Visualforce page. For known limitations of this approach, see Lightning Out Considerations and Limitations.

See Also

- Use Components in Visualforce Pages (*Lightning Web Components Dev Guide*)
- Quick Start: Lightning Web Components (Trailhead project)

CHAPTER 10 React Native Development

In this chapter ...

React Native is a third-party framework that lets you access native UI elements directly with JavaScript, style sheets, and markup. You can combine this technology with special Mobile SDK native modules for rapid development using native resources.

Since its inception, Mobile SDK has supported two types of mobile apps:

- **Native apps** provide the best user experience and performance. However, you have to use a different development technology for each mobile platform you support.
- **Hybrid apps** let you share your JavaScript and style sheets code across platforms, but the generic underlying web view can comprise the user experience.

In Mobile SDK 4.0 and later, you have a third option: React Native. React Native couples the cross-platform advantages of JavaScript development with the platform-specific "look and feel" of a native app. At the same time, the developer experience matches the style and simplicity of hybrid development.

- You use flexible, widely known web technologies (JavaScript, style sheets, and markup) for layout and styling.
- No need to recompile to check your code updates. You simply refresh the browser to see your changes.
- To debug, you use your favorite browser's developer tools.
- All views are rendered natively, so your customers get the user experience of a native app.

Mobile SDK 11.1 uses React Native 0.70.14. You can find React Native 0.70.14 source code and documentation at github.com/facebook/react-native/releases/ under the 0.70.14 tag.

Getting Started

React Native requires some common Mobile SDK components and at least one native development environment—iOS or Android. On a macOS machine, you can develop both iOS and Android versions of your app. On Windows, you're limited to developing for Android.

In Mobile SDK 9.0, you have the option of developing your React Native app using plain JavaScript (ES2015) or TypeScript. TypeScript gives you compile-time static type checking and custom types in a standard JavaScript environment. To learn more, see TypeScript Documentation.

Your best bet for getting started is the React Native Trailhead module. See you back here afterwards.

Set Up Your React Native Development Environment

Mobile SDK provides forcereact, a command-line script for installing React Native and Mobile SDK libraries, and creating projects from template apps. To support this tool, install the following packages.

1. Install git.
   a. To check if git is already installed, at the operating system command prompt type `git version` and press Return.

   b. If you get a "command not found" error message, download and install the git package for your operating system at git-scm.com/downloads.

2. Install Node.js and npm.
   a. To check if these tools are already installed, at the command prompt type `npm -v` and press Return.
   b. If you get a "command not found" error message, download and install the Node.js package for your operating system at https://nodejs.org/en/.

3. Install yarn.

```
[sudo] npm install -g yarn
```

   For more information, see **Getting Started** > **Installation** at https://yarnpkg.com/

4. Install the TypeScript compiler.

```
[sudo] npm install -g typescript
```

   For more information, see **Download** at https://www.typescriptlang.org.

   Warning: Don't use `npm install -g tsc`. This variation installs an incompatible version of TypeScript that causes forcereact to fail.

5. Install forcereact.

```
[sudo] yarn global add forcereact
```

Important: To satisfy Google Play requirements, Mobile SDK 7.2 upgrades its React Native component to a version that supports 64-bit builds. Starting **August 1, 2019**, *new* and *updated* Mobile SDK React Native apps that will be published on the Google Play store must be built with Mobile SDK 7.2 or later. See https://android-developers.googleblog.com/2019/01/get-your-apps-ready-for-64-bit.html?m=1 for full details.

Creating a React Native Project with Forcereact

After you've successfully installed a native Mobile SDK environment, you can begin creating React Native apps.

To create an app, use forcereact in a terminal window or at a Windows command prompt. The forcereact utility gives you two ways to create your app.

- Specify the type of application you want, along with basic configuration data.

  OR

- Use an existing Mobile SDK app as a template. You still provide the basic configuration data.

In Mobile SDK 9.0, forcereact adds an app type option to support TypeScript: `react_native_typescript`. This type becomes the default for the `forcereact create` command. To instead use standard JavaScript, you specify the `react_native` app type.

You can use forcereact in interactive mode with command-line prompts, or in scripted mode with the parameterized command-line version.

Using Forcereact Interactively

To enter application options interactively at a command prompt, type *`forcereact create`*. The forcereact utility then prompts you for each configuration option.

Using Forcereact with Command-Line Options

If you prefer, you can specify forcereact parameters directly at the command line. To see usage information, type *`forcereact`* without arguments. The list of available options displays:

```
$ forcereact

forcereact: Tool for building a React Native mobile application
            using Salesforce Mobile SDK

Usage:

# create a React Native mobile application
forcereact create
    --platform=comma-separated list of platforms (ios, android)
    [--apptype=application type (react_native_typescript or react_native,
        leave empty for react_native_typescript)]
    --appname=application name
    --packagename=app package identifier (e.g. com.mycompany.myapp)
    --organization=organization name (your company's/organization's name)
    [--outputdir=output directory (leave empty for current directory)]
```

Using this information, type *`forcereact create`*, followed by your options and values. For example, to create a React Native app that supports TypeScript:

```
$ forcereact create
--platform=ios,android
--appname=CoolReact
--packagename=com.test.my_new_app
```

```
--organization="Acme Widgets, Inc."
--outputdir=CoolReact
```

Specifying a Template

forcereact createWithTemplate is identical to *forcereact create* except that it also asks for a template repo URI. You set this path to point to any repo directory that contains a Mobile SDK app that can be used as a template. Your template app can be any supported Mobile SDK app type—`react_native` or `react_native_typescript`. The force script changes the template's identifiers and configuration to match the values you provide for the other parameters.

Before you use `createWithTemplate`, it's helpful to know which templates are available. To find out, type *forcereact listtemplates*. This command prints a list of templates provided by Mobile SDK. Each listing includes a brief description of the template and its GitHub URI. For example:

```
Available templates:

1) Basic React Native application:
    forcereact createwithtemplate --templaterepouri=ReactNativeTemplate
2) Basic React Native application that uses deferred login
    forcereact createwithtemplate --templaterepouri=ReactNativeDeferredTemplate
3) Basic React Native application written in TypeScript
    forcereact createwithtemplate --templaterepouri=ReactNativeTypeScriptTemplate
4) Sample application using MobileSync data framework
   forcereact createwithtemplate --templaterepouri=MobileSyncExplorerReactNative
```

After you've found a template's URI, you can plug it into the forcereact command line. Here's command-line usage information for creating a TypeScript project with `forcereact createWithTemplate`:

```
forcereact createWithTemplate
    --platform=Comma separated platforms (ios, android)
    --templaterepouri=Template repo URI
    --appname=Application Name
    --packagename=App Package Identifier (e.g. com.mycompany.myapp)
    --organization=Organization Name (Your company's/organization's name)
    [--outputdir=Output Directory (Leave empty for current directory)]
```

For any template in the `SalesforceMobileSDK-Templates` repo, you can drop the path for `templaterepouri`—just the template name will do. For example, consider the following command-line call:

```
forcereact createWithTemplate
--platform=ios,android
--templaterepouri=MobileSyncExplorerReactNative
--appname=MyReact
--packagename=com.mycompany.react
--organization="Acme Software, Inc."
--outputdir=""
```

This call creates a React Native app in the current directory that supports both iOS and Android. It uses the same source code and resources as the MobileSyncExplorerReactNative sample app. Forcereact changes the app name to "MyReact" throughout the project.

Build and Run Your App with Android Studio

For React Native, Mobile SDK supports both `npm` and `yarn` for package management. When you see both options, the choice is yours.

1. In a Terminal window or at Windows command prompt, change to your project's root directory.

   ```
   cd <my_project_root>
   ```

2. Run the following command:
   - `npm start`

     OR
   - `yarn start`
3. Open your project in Android Studio.
   - From the Welcome screen, click **Import Project (Eclipse ADT, Gradle, etc.)**.

     OR
   - From the File menu, click **File** > **New** > **Import Project...**.
4. Browse to your `<project_ root>/android/` directory and click **OK**.

   Android Studio automatically builds your workspace. This process can take several minutes. When the status bar reports "Gradle build successful", you're ready to run the project.
5. Click **Run 'app'**, or press `SHIFT+F10`.

   Android Studio launches your app in the emulator or on your connected Android device.

Build and Run Your App in Xcode

1. In a Terminal window, change to your project's root directory.

   ```
   cd <my_project_root>
   ```

2. Type `npm start`, then press Return.
3. In Xcode, open `<project_ root>/ios/<project_name>.xcworkspace`.
4. Click **Run**.

   Xcode launches your app in the simulator or on your connected iOS device.

How the Forcereact Script Generates New Apps

- The script downloads templates at runtime from a GitHub repo. For the `forcereact create` command, the script uses the default templates in the SalesforceMobileSDK-Templates GitHub repo.
- The script uses npm to download React Native dependencies.
- The script uses git (Android) or CocoaPods (iOS) to download Mobile SDK libraries.

Using TypeScript in React Native Projects

TypeScript brings useful advantages to React Native apps. Not only does it help you write safer code—it also coexists seamlessly with vanilla JavaScript. You can use as much or as little TypeScript as you like.

To demonstrate TypeScript usage, Mobile SDK provides a new template app. You can view or download the new template at the ReactNativeTypeScriptTemplate directory of the SalesforceMobileSDK-Templates GitHub repo.

Source code for TypeScript apps can reside in `*.ts`, `*.tsx`, or `*.js` files. For example, in the new template app, `app.js` becomes `app.tsx`. A comparison between the old and new files demonstrates how unobtrusive TypeScript is.

- Imports are the same.
- Code changes for new types are minimal.

In `app.tsx`, the template adds the following custom types to the original JavaScript code:

```
interface Response {
    records: Record[]
}

interface Record {
    Id: String,
    Name: String
}

interface Props {
}

interface State {
    data : Record[]
}
```

The `Record` type is used internally in the `Response` interface. The new template applies the other three types to parameters of the original script. For example, `Props` and `State` apply to the constructor:

```
class ContactListScreen extends React.Component<Props, State> {
    constructor(props:Props) {
        super(props);
        this.state = {data: []};
    }
```

The `Response` type secures the response received from the SOQL query:

```
(response:Response) => that.setState({data: response.records}),
    (error) => console.log('Failed to query:' + error)
```

If Salesforce passes data to `response` that conflicts with definitions in the `Response` type, you get a compile-time error. However, the `Response` interface doesn't disallow or inspect objects it doesn't define. Thus, although `Response` specifies only two fields—`Id` and `Name`—accessing `response.records` passes muster as plain JavaScript.

Using Mobile SDK Components in React Native Apps

React Native apps access the same Mobile SDK libraries as Mobile SDK native apps. For React Native, Mobile SDK provides JavaScript components, or *bridges*, that execute your JavaScript code as native Mobile SDK instructions.

In React Native, you access Mobile SDK functionality through the following native bridges:

- `react.force.oauth.js`
- `react.force.net.js`
- `react.force.smartstore.js`

- `react.force.mobilesync.js`

To use these bridges, add an import statement in your JavaScript code. The following example imports all four bridges.

```
import {oauth, net, smartstore, mobilesync} from 'react-native-force';
```

React native apps built with forcereact specify the `react-native-force` source path in the `package.json` file:

```
"react-native-force": "https://github.com/forcedotcom/SalesforceMobileSDK-ReactNative.git"
```

Note: You can't use the `force.js` library with React Native.

Mobile SDK Native Modules for React Native Apps

Mobile SDK provides native modules for React Native that serve as JavaScript bridges to native Mobile SDK functionality.

OAuth

The OAuth bridge is similar to the OAuth plugin for Cordova.

Usage

```
import {oauth} from 'react-native-force';
```

Types and Methods

See react.force.oauth.ts in the SalesforceMobileSDK-ReactNative GitHub repo.

Network

The Network bridge is similar to the force.js library for hybrid apps.

Usage

```
import {net} from 'react-native-force';
```

Types and Methods

See react.force.net.ts in the SalesforceMobileSDK-ReactNative GitHub repo.

SmartStore

The SmartStore bridge is similar to the SmartStore plugin for Cordova. Unlike the plugin, however, first arguments aren't optional in React Native.

Usage

```
import {smartstore} from 'react-native-force';
```

Types and Methods

See react.force.smartstore.ts in the SalesforceMobileSDK-ReactNative GitHub repo.

Mobile Sync

The Mobile Sync bridge is similar to the Mobile Sync plugin for Cordova. Unlike the plugin, however, first arguments aren't optional in React Native.

Usage

```
import {mobilesync} from 'react-native-force';
```

Types and Methods

See react.force.mobilesync.ts in the SalesforceMobileSDK-ReactNative GitHub repo.

Note: Handling of field lists for "sync up" operations changed in Mobile SDK 5.1. See Mobile Sync Plugin Methods for a description of the JavaScript `syncUp()` method.

Mobile SDK Sample App Using React Native

The best way to get up-to-speed on React Native in Mobile SDK is to study the sample code.

Mobile SDK provides four implementations of the MobileSyncExplorer application, including a React Native version. To use MobileSyncExplorerReactNative, follow the instructions in the MobileSyncExplorerReactNative README.md file.

Here are a few notes about the MobileSyncExplorerReactNative files.

Table 1: Key Folder and Files

Path	Description
`README.md`	Instructions to get started
`installandroid.js`	Use this script to install the Android sample. See `README.md` for details.
`installios.js`	Use this script to install the iOS sample. See `README.md` for details.
`ios`	The iOS application
`android`	The Android application
`js`	The JavaScript source files for the application
`index.js`	App start page

Table 2: React Components

File	Component	Description
`js/events.js`		Event model
`js/App.js`	MobileSyncExplorerReactNative	Root component (the entire application) (iOS and Android)
`js/SearchScreen.js`	SearchScreen	Search screen (iOS and Android)
`jsContactScreen.js`	ContactScreen	Used for viewing and editing a single contact (iOS and Android)

File	Component	Description
`js/ContactCell.js`	ContactCell	A single row in the list of results in the search screen (iOS and Android)
`js/ContactBadge.js`	ContactBadge	Colored circle with initials used in the search results screen (iOS and Android)
js/Field.js	Field	A field name and value used in the contact screen (iOS and Android)
js/StoreMgr.js	StoreMgr	Interacts with SmartStore and the server (via Mobile Sync).
js/NavImgButton.js	NavImgButton	Navigation Bar button

Table 3: Platform-Specific Native Projects

File	Description
`android/`	Android native project
`ios/`	iOS native project

Note: Most components are shared between iOS and Android. However, some components are platform-specific.

Defer Login

Apps built with early versions of React Native for Mobile SDK always present a Salesforce login screen at startup. Sometimes, however, these apps can benefit from deferring authentication until some later point. Beginning with React Native for Mobile SDK 4.2, you can defer login to any logical place in your app.

Deferred login implementation is a two-step process:

1. In your iOS or Android native container app, you call Mobile SDK native methods to disable authentication at startup.
2. In your React code, you call a Mobile SDK JavaScript function at the point where you plan to initiate authentication.

Mobile SDK provides a React Native template app specifically to demonstrate deferred login: ReactNativeDeferredTemplate. To create an app based on this template, use the `forcereact createwithtemplate` command, as follows:

```
$ forcereact createwithtemplate
Enter the target platform(s) separated by commas (ios, android): ios,android
Enter URI of repo containing template application or a Mobile SDK template name:
ReactNativeDeferredTemplate
Enter your application name: MyDeferred
...
```

Read on for the implementation details.

Step1: Disable Login at Startup

iOS (Objective-C):

To disable the Salesforce login screen from appearing at startup:

1. In your project's `bootconfig.plist` file, set `shouldAuthenticate` to `false`.

```
<dict>
    <key>remoteAccessConsumerKey</key>
    <string>3REW9Iu66....</string>
    <key>oauthRedirectURI</key>
    <string>testsfdc:///mobilesdk/detect/oauth/done</string>
    <key>oauthScopes</key>
    <array>
        <string>web</string>
        <string>api</string>
    </array>
    <key>shouldAuthenticate</key>
    <false/>
</dict>
```

Android (Java):

By default, the Salesforce login screen appears at startup. To disable this behavior, override the `shouldAuthenticate()` method in your `MainActivity` class (or whichever class subclasses `SalesforceReactActivity`), as follows:

```
@Override
public boolean shouldAuthenticate() {
    return false;
}
```

Step 2: Initiate Authentication in React (JavaScript)

To initiate the authentication process, call the following `oauth` function:

```
function authenticate(success, fail)
```

This function takes two arguments: a success callback function and a failure callback function. If authentication fails, your failure callback is invoked. If authentication succeeds, your success callback is invoked with a dictionary containing the following keys:

- `accessToken`
- `refreshToken`
- `clientId`
- `userId`
- `orgId`
- `loginUrl`
- `instanceUrl`
- `userAgent`
- `communityId`
- `communityUrl`

Upload Binary Content

You can upload binary content to any `force.com` endpoint that supports the binary upload feature.

The `sendRequest()` method in `react.force.net.js` has a new optional parameter named `fileParams`.

```
function sendRequest(endPoint, path, successCB, errorCB, method, payload, headerParams,
fileParams)
```

This parameter expects the following form:

```
{
    <fileParamNameInPost>: // value depends on the endpoint
    {
        fileMimeType:<someMimeType>,
        fileUrl:<fileUrl>, // url to file to upload
        fileName:<fileNameForPost>
    }
}
```

For example:

```
{
    fileUpload:
    {
        fileMimeType:'image/jpeg',
        fileUrl:localPhotoUrl,
        fileName:'pic.jpg'
    }
}
```

Example: The github.com/wmathurin/MyUserPicReactNative sample app demonstrates binary upload. This sample allows you to change your profile picture. Binary upload of the new pic happens in the `uploadPhoto()` function of the `UserPic.js` file.

Here's the sample's `sendRequest()` call in the `getUserInfo()` function:

```
getUserInfo(callback) {
    forceClient.sendRequest('/services/data',
        '/v36.0/connect/user-profiles/' + this.state.userId + '/photo',
        (response) => {
            callback(response);
        },
        (error) => {
            console.log('Failed to upload user photo:' + error);
        },
        'POST',
        {},
        {'X-Connect-Bearer-Urls': 'true'},
        {fileUpload:
            {
                fileUrl:localPhotoUrl,
                fileMimeType:'image/jpeg',
                fileName:'pic.jpg'
            }
        }
    );
},
```

CHAPTER 11 Offline Management

In this chapter ...

Salesforce Mobile SDK provides two modules that help you store and synchronize data for offline use:

- SmartStore is an offline storage solution that saves encrypted Salesforce data to the device. It is built on SqlLite and uses SqlCipher to encrypt customer data. SmartStore can handle flexible, evolving data models. Unlike a traditional database, it allows records with varying shapes. Apps that use SmartStore define custom indexed tables, known as "soups", to organize and manage their data. To populate soups, you use Mobile SDK APIs to request data from Salesforce endpoints, and then call SmartStore methods to upsert the returned records. SmartStore provides its own query language, Smart SQL, for retrieving data from the store. SmartStore also supports full text search and is available on native, hybrid, and React Native platforms.
- Mobile Sync provides a mechanism for easily synchronizing Salesforce records in the cloud with local records in SmartStore. Mobile Sync APIs call the Salesforce API on your behalf to retrieve or upsert Salesforce data and populate your SmartStore soups. When it's time to upsert records to Salesforce, Mobile Sync gives you fine-grained control over the sync process. Mobile Sync uses SmartStore as its default data store.

For example, let's say a sales agent on customer visits is traveling through areas with low internet connectivity. Your mobile app can:

- Use Mobile SDK REST APIs or the Mobile Sync `reSync()` method to preload customer data from the Salesforce cloud into SmartStore.
- During travel, save edits and updates to local SmartStore soups, regardless of the device's connection status.
- When the device comes back online, or at predetermined intervals, use Mobile Sync APIs to sync data between the local store and the Salesforce cloud.

Important: SmartSync Data Framework has been renamed "Salesforce Mobile Sync". Accordingly, all code uses of "SmartSync" in the SDK have changed to "MobileSync", and module names have changed as well. To upgrade from Mobile SDK 7.3 or earlier, update your existing projects to reflect these changes.

The name "SmartStore" does not change.

Using SmartStore to Securely Store Offline Data

Mobile devices can lose connection at any time, and environments such as hospitals and airplanes often prohibit connectivity. To handle these situations, it's important that your mobile apps continue to function when they go offline.

Mobile SDK provides SmartStore, a multithreaded, secure solution for offline storage on mobile devices. With SmartStore, your customers can continue working with data in a secure environment even when the device loses connectivity. When you couple SmartStore with Mobile Sync, you can easily keep local SmartStore data in sync with the Salesforce server when connectivity resumes.

IN THIS SECTION:

About SmartStore

SmartStore provides the primary features of non-relational desktop databases—data segmentation, indexing, querying—along with caching for offline storage.

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

What's New in SmartStore

Smart SQL no longer requires index paths for all fields referenced in SELECT or WHERE clauses This improvement doesn't extend to soups that use external storage.

About Data Caching

To provide offline synchronization and conflict resolution services, SmartStore uses StoreCache, a Mobile SDK caching mechanism. We recommend that you use StoreCache to manage operations on Salesforce data.

Note: Pure HTML5 apps store offline information in a browser cache. Browser caching isn't part of Mobile SDK, and we don't document it here. SmartStore uses storage functionality on the device. This strategy requires a native or hybrid development path.

About the Sample Code

Objective-C code snippets in this chapter use Account and Opportunity objects, which are predefined in every Salesforce organization. Accounts and opportunities are linked through a master-detail relationship. An account can be the master for more than one opportunity.

IN THIS SECTION:

SEE ALSO:

Using StoreCache For Offline Caching

Conflict Detection

Smart SQL Queries

SmartStore Soups

SmartStore soups let you partition your offline content.

SmartStore stores offline data in logical collections known as *soups*. A SmartStore soup represents a single table in the underlying SQLite database, or *store*, and typically maps to a standard or custom Salesforce object. Soups contain *soup elements*. Each element is a JSON object that mirrors a single database row. To streamline data access, you define indexes for each soup. You use these indexes to query the soup with either SmartStore helper methods or SmartStore's Smart SQL query language. SmartStore indexes also make your life easier by supporting full-text search queries.

It's helpful to think of soups as tables, and stores as databases. You can define as many soups as you like in an application. As self-contained data sets, soups don't have predefined relationships to each other, but you can use Smart SQL joins to query across them. Also, in native apps you can write to multiple soups within a transaction.

Warning: SmartStore data is volatile. In most cases, its lifespan is tied to the authenticated user and to OAuth token states. When the user logs out of the app, SmartStore deletes all soup data associated with that user. Similarly, when the OAuth refresh token is revoked or expires, the user's app state is reset, and all data in SmartStore is purged. When designing your app, consider the volatility of SmartStore data, especially if your organization sets a short lifetime for the refresh token.

SmartStore Stores

SmartStore puts encrypted soup data in an underlying system database known as the *store*. The store is where all soup data is stored, encrypted, related, and indexed. If the device loses connectivity, the user can continue to work on data in the store until the Salesforce cloud is again accessible.

When you initialize SmartStore, you specify the name of a store to open. You assign a custom name or use a standard name, such as `kDefaultSmartStoreName` in iOS native apps, to define the store. Named stores are user-specific—like soups, the store persists only while the user's session remains valid.

In a traditional SmartStore session, all soups reference, organize, and manipulate content from a single store. Single-store configuration is the best choice for many apps. However, if an app queries large quantities of data from many objects, performance can begin to degrade. To avoid slower response time, you can create multiple named stores and partition your data between them. For example, if your app defines tasks that operate on clear-cut domains of Salesforce data, you can create a store for each task. Runtime access to a smaller store can make a big difference in user satisfaction.

Some use cases require a store that isn't tied to a user's login credentials and can persist between user and app sessions. SmartStore accommodates this requirement by supporting *global stores*. Global stores are also named stores, but you create and remove them through a different set of APIs.

SEE ALSO:

- Using Global SmartStore
- Creating and Accessing User-based Stores

SmartStore Data Types

Like any database, SmartStore defines a set of data types that you use to create soups. SmartStore data types mirror the underlying SQLite database.

SmartStore supports the following data types for declaring *index specs*. In a SmartStore soup definition, an index spec defines the data type that SmartStore expects to find in the given field.

Type	Description
`integer`	Signed integer, stored in 4 bytes (SDK 2.1 and earlier) or 8 bytes (SDK 2.2 and later)
`floating`	Floating point value, stored as an 8-byte IEEE floating point number
`string`	Text string, stored with database encoding (UTF-8)
`full_text`	String that supports full-text searching
`JSON1`	Index type based on the SQLite JSON1 extension. Can be used in place of integer, floating, and string types. Behaves identically to those types of index specs, except that JSON1 does not support index paths that traverse arrays.

IN THIS SECTION:

SEE ALSO:

Using Arrays in Index Paths

Date Representation

SmartStore does not define a date/time data type. When you create index specs for date/time fields, choose a SmartStore type that matches the format of your JSON input. For example, Salesforce sends dates as strings, so always use a string index spec for Salesforce date fields. To choose an index type for non-Salesforce or custom date fields, consult the following table.

Type	Format As	Description
`string`	An ISO 8601 string	"YYYY-MM-DD HH:MM:SS.SSS"
`floating`	A Julian day number	The number of days since noon in Greenwich on November 24, 4714 BC according to the proleptic Gregorian calendar. This value can include partial days that are expressed as decimal fractions.
`integer`	Unix time	The number of seconds since 1970-01-01 00:00:00 UTC

Enabling SmartStore in Hybrid and Native Apps

To use SmartStore in hybrid Android apps, you perform a few extra steps.

Hybrid apps access SmartStore through JavaScript. To enable offline access in a hybrid app, your Visualforce or HTML page must include the `cordova.js` library file.

In forceios and forcedroid native apps, SmartStore is always included.

SEE ALSO:

Creating an Android Project with Forcedroid

Creating an iOS Project with Forceios

Adding SmartStore to Existing Android Apps

Hybrid projects created with Mobile SDK 4.0 or later automatically include SmartStore. If you used Mobile SDK 4.0+ to create an Android native project without SmartStore, you can easily add it later.

To add SmartStore to an existing native Android project (Mobile SDK 4.0 or later):

1. Add the SmartStore library project to your project. In Android Studio, open your project's `build.gradle` file and add a compile directive for `:libs:SmartStore` in the `dependencies` section. If the `dependencies` section doesn't exist, create it.

```
dependencies {
...
```

```
  compile project(':libs:SmartStore')
}
```

2. In your <*projectname*>`App.java` file, import the `SmartStoreSDKManager` class instead of `SalesforceSDKManager`. Replace this statement:

```
import com.salesforce.androidsdk.
    app.SalesforceSDKManager
```

with this one:

```
import com.salesforce.androidsdk.smartstore.app.SmartStoreSDKManager
```

3. In your <*projectname*>`App.java` file, change your App class to extend the `SmartStoreSDKManager` class rather than `SalesforceSDKManager`.

Note:

1. To add SmartStore to apps created with Mobile SDK 3.x or earlier, begin by upgrading to the latest version of Mobile SDK.
2. The SmartStore plugin, `com.salesforce.plugin.smartstore.client`, uses promises internally.

   Mobile SDK promised-based APIs include:

   - `force+promise.js`
   - The `smartstoreclient` Cordova plugin (`com.salesforce.plugin.smartstore.client`)
   - `mobilesync.js`

SEE ALSO:

Migrating from the Previous Release

Creating and Accessing User-based Stores

When an app initializes SmartStore, it creates an instance of a store. It then uses the store to register and populate soups and manipulate soup data. For a user-based store, SmartStore manages the store's life cycle—you don't need to think about cleaning up after the user's session ends. For global stores, though, your app is responsible for deleting the store's data when the app terminates.

Android Native Apps

Android requires you to first get an instance of `SmartStoreSDKManager` which you then use to create stores.

```
SmartStoreSDKManager sdkManager =
    SmartStoreSDKManager.getInstance();

SmartStore smartStore = sdkManager.getSmartStore(); // Creates a default store for the
current user
```

A call to `SmartStoreSDKManager.getSmartStore()` without arguments always accesses the default anonymous store. To create a named user-based store, call the following method.

```
public SmartStore getSmartStore(String dbNamePrefix, UserAccount account, String communityId)
```

Both `account` and `communityId` can be null. You can call these methods as many times as necessary to create additional stores.

iOS Native Apps

For creating stores, iOS provides the `sharedStoreWithName:` class message.

```
- (SFSmartStore *)store
{
    return [SFSmartStore sharedStoreWithName:kDefaultSmartStoreName]; // Creates a default
 store for the current user
}
```

In Swift:

```
var store = SmartStore.shared(withName: storeName)
```

You can create a store with a custom name by passing in any string other than `kDefaultSmartStoreName`. You can call this method as many times as necessary to create additional stores.

Hybrid Apps

In hybrid apps, you access user-based stores and global stores the same way. Rather than creating stores explicitly, you automatically create stores that don't already exist when you call `registerSoup()`. To use a named store—for subsequent direct references, for example—you call this function with a `StoreConfig` object as the first argument. This function object takes a store name and a Boolean value that indicates whether the store is global.

```
var StoreConfig = function (storeName, isGlobalStore) {
    this.storeName = storeName;
    this.isGlobalStore = isGlobalStore;
};
```

You can pass this object as the optional first argument to most soup functions. If used, the `StoreConfig` object configures the execution context. Either `storeName` or `isGlobalStore` can be optional—you can specify one or both. SmartStore evaluates `StoreConfig` objects as follows:

- If `storeName` is not specified, `this.storeName` is set to the SmartStore default store name.
- If `isGlobalStore` is not specified, `this.isGlobalStore` is set to `false`.
- Store names aren't necessarily unique. A single store name can be used twice—once for a user-based store, and once for a global store.
- If you provide a store name that doesn't exist in the space indicated by your `isGlobalStore` setting, SmartStore creates it.

The following example creates a user-based store named "Store1" that contains the `soupName` soup.

```
navigator.smartstore.registerSoup({storeName: "Store1", isGlobalStore:false}, soupName,
    indexSpecs, successCallback, errorCallback)
```

You can call `registerSoup()` with as many different soup names as necessary. If you call a soup function without passing in `StoreConfig`, SmartStore always performs the operation on the default user-based (non-global) store. This behavior applies even if you've created named stores. The following example creates a soup named `soupName`, with the provided index specs, in the current user's default store.

```
var sfSmartstore = function() {
    return cordova.require("com.salesforce.plugin.smartstore");
```

```
};
sfSmartstore().registerSoup(soupName, indexSpecs, successCallback, errorCallback);
```

SEE ALSO:

SmartStore Stores

Using Global SmartStore

Although you usually tie a SmartStore instance to a specific customer's credentials, you can also access a global instance for special requirements.

Under certain circumstances, some applications require access to a SmartStore instance that is not tied to Salesforce authentication. This situation can occur in apps that store application state or other data that does not depend on a Salesforce user, organization, or community. To address this situation, Mobile SDK supports global stores that persists beyond the app's life cycle.

Data stored in global stores does not depend on user authentication and therefore is not deleted at logout. Since a global store remains intact after logout, you are responsible for deleting its data when the app exits. Mobile SDK provides APIs for this purpose.

Important: Do not store user-specific data in global SmartStore. Doing so violates Mobile SDK security requirements because user data can persist after the user logs out.

Android APIs

In Android, you access global SmartStore through an instance of `SmartStoreSDKManager`.

- ```
  public SmartStore getGlobalSmartStore(String dbName)
  ```

  Returns a global SmartStore instance with the specified database name. You can set `dbName` to any string other than "smartstore". Set `dbName` to null to use the default global SmartStore database.

- ```
  public boolean hasGlobalSmartStore(String dbName)
  ```

  Checks if a global SmartStore instance exists with the specified database name. Set `dbName` to null to verify the existence of the default global SmartStore.

- ```
  public void removeGlobalSmartStore(String dbName)
  ```

  Deletes the specified global SmartStore database. You can set this name to any string other than "smartstore". Set `dbName` to null to remove the default global SmartStore.

iOS APIs

In iOS, you access global SmartStore through an instance of `SFSmartStore`.

- **Objective-C:**

  ```
  + (id)sharedGlobalStoreWithName:(NSString *)storeName
  ```

  Swift:

  ```
  var gstore = SmartStore.sharedGlobal(withName: storeName)
  ```

  Returns a global SmartStore instance with the specified database name. You can set `storeName` to any string other than "defaultStore". Set `storeName` to `kDefaultSmartStoreName` to use the default global SmartStore.
  ```

- **Objective-C:**

  ```
  + (void)removeSharedGlobalStoreWithName:(NSString *)storeName
  ```

  Swift:

  ```
  SmartStore.removeSharedGlobal(withName: storeName)
  ```

  Deletes the specified global SmartStore database. You can set `storeName` to any string other than "defaultStore". Set `storeName` to `kDefaultSmartStoreName` to use the default global SmartStore.

Hybrid APIs

Most SmartStore JavaScript soup methods take an optional first argument that specifies whether to use global SmartStore. This argument can be a Boolean value or a `StoreConfig` object. If this argument is absent, Mobile SDK uses the default user store.

For example:

```
querySoup([isGlobalStore, ]soupName, querySpec,
    successCB, errorCB);
querySoup([storeConfig, ]soupName, querySpec,
    successCB, errorCB);
```

SmartStore defines the following functions for removing stores. Each function takes success and error callbacks. The `removeStore()` function also requires either a `StoreConfig` object that specifies the store name, or just the store name as a string.

```
removeStore(storeConfig,successCB, errorCB)
removeAllGlobalStores(successCB, errorCB)
removeAllStores(successCB, errorCB)
```

SEE ALSO:

- SmartStore Stores
- Managing Stores
- Creating and Accessing User-based Stores

Registering Soups with Configuration Files

Beginning with Mobile SDK 6.0SmartStore lets you define soup structures through configuration files rather than code. Since all platforms and app types use the same configuration files, you can describe all your soups in a single file. You can then compile that file into any Mobile SDK project.

To register a soup, you provide a soup name and a list of one or more index specifications.

You index a soup on one or more fields found in its entries. SmartStore makes sure that these indexes reflect any insert, update, and delete operations. Always specify at least one index field when registering a soup. For example, if you are using the soup as a simple key-value store, use a single index specification with a string type.

Overview

SmartStore configuration files use JSON objects to express soup definitions. The JSON schema for configuration files is the same for all app types and platforms. Hybrid apps load the configuration files automatically, while other apps load them with a single line of code. To keep the mechanism simple, Mobile SDK enforces the following file naming conventions:

- To define soups for the default global store, provide a file named `globalstore.json`.
- To define soups for the default user store, provide a file named `userstore.json`.

Configuration files can define soups only in the default global store and default user store. For named stores, you register soups through code. You can't use configuration files to set up externally stored soups.

Note:

- Configuration files are intended for initial setup only. You can't change existing soups by revising the JSON file and reloading it at runtime. Instead, use SmartStore methods such as `alterSoup()`. See Managing Soups.
- If a configuration file defines a soup that exists, Mobile SDK ignores the configuration file. In this case, you can set up and manage your soups only through code.

Configuration File Format

The JSON format is self-evident as illustrated in the following example.

```
{  "soups": [
    {
      "soupName": "soup1",
      "indexes": [
        { "path": "stringField1", "type": "string"},
        { "path": "integerField1", "type": "integer"},
        { "path": "floatingField1", "type": "floating"},
        { "path": "json1Field1", "type": "json1"},
        { "path": "ftsField1", "type": "full_text"}
      ]
    },
    {
      "soupName": "soup2",
      "indexes": [
        { "path": "stringField2", "type": "string"},
        { "path": "integerField2", "type": "integer"},
        { "path": "floatingField2", "type": "floating"},
        { "path": "json1Field2", "type": "json1"},
        { "path": "ftsField2", "type": "full_text"}
      ]
    }
  ]
}
```

For Mobile Sync compatibility, configuration files also require indexes on some system fields. See Preparing Soups for Mobile Sync.

Configuration File Locations

Configuration file placement varies according to app type and platform. Mobile SDK looks for configuration files in the following locations:

iOS (Native and React Native)
Under `/` in the Resources bundle

Android (Native and React Native)
In the `/res/raw` project folder

Hybrid
In your Cordova project, do the following:

1. Place the configuration file in the top-level `www/` folder.
2. In the top-level project directory, run: *`cordova prepare`*

Loading SmartStore Configuration Files in Native Apps

SmartStore and its companion feature Mobile Sync require a special SDK manager object. For example, to use SmartStore or Mobile Sync in iOS, initialize the SDK by calling `MobileSyncSDKManager.initializeSDK()` rather than `SalesforceSDKManager.initializeSDK()`.

If you're not using Mobile Sync, you can call `SmartStoreSDKManager.initializeSDK()`. However, such cases are rare.

In native and React Native apps, you load your JSON configuration file by calling a loading method. Make this call in your app initialization code after the customer successfully logs in. For example, in iOS, make this call in the block you pass to loginIfRequired. Call these methods only if you're using a globalstore.json or userstore.json file instead of code to configure SmartStore. Do not call these loading methods more than once. In hybrid apps that include them, SmartStore configuration files are loaded automatically.

To load a soup configuration file, call the loader method for the store you're targeting. Load this file before calling other SmartStore methods.

iOS (Native and React Native)

Load a soup configuration file by calling the appropriate method on the `MobileSyncSDKManager` object.

Load a Default User Store

Swift

```
// In the AppDelegate class:
override init() {
        super.init()
        MobileSyncSDKManager.initializeSDK()
...

// Load config files in the block you pass to loginIfRequired()
func application(_ application: UIApplication, didFinishLaunchingWithOptions
    launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

    self.window = UIWindow(frame: UIScreen.main.bounds)
    self.initializeAppViewState()
    // ...

    AuthHelper.loginIfRequired {
        self.setupRootViewController()
        MobileSyncSDKManager.shared.setupUserStoreFromDefaultConfig()
    }
...
```

Objective-C

```
// In the AppDelegate class:
- (instancetype)init
{
    self = [super init];
    if (self) {
        [MobileSyncSDKManager initializeSDK];
...
```

```
// Load config files in the block you pass to loginIfRequired()
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
    [self initializeAppViewState];
...

    [SFSDKAuthHelper loginIfRequired:^{
        [self setupRootViewController];
        [[MobileSyncSDKManager sharedManager] setupUserStoreFromDefaultConfig];
    }];
...
```

Load a Default Global Store

Swift

```
// In the AppDelegate class:
override init() {
        super.init()
        MobileSyncSDKManager.initializeSDK()
...

// Load config files in the block you pass to loginIfRequired()
func application(_ application: UIApplication, didFinishLaunchingWithOptions
    launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    self.window = UIWindow(frame: UIScreen.main.bounds)
    self.initializeAppViewState()
    // ...

    AuthHelper.loginIfRequired {
        self.setupRootViewController()
        MobileSyncSDKManager.shared.setupGlobalStoreFromDefaultConfig()
    }
...
```

Objective-C

```
// In the AppDelegate class:
- (instancetype)init
{
    self = [super init];
    if (self) {
        [MobileSyncSDKManager initializeSDK];
...

// Load config files in the block you pass to loginIfRequired()
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
    [self initializeAppViewState];
...

    [SFSDKAuthHelper loginIfRequired:^{
```

```
        [self setupRootViewController];
        [[MobileSyncSDKManager sharedManager] setupGlobalStoreFromDefaultConfig];
    }];
...
```

Android (Native and React Native)

Load a soup configuration file by calling the appropriate method on the `MobileSyncSDKManager` object.

Load a Default User Store

```
public class MainApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        MobileSyncSDKManager.initNative(getApplicationContext(), MainActivity.class);
        MobileSyncSDKManager.getInstance().setupUserStoreFromDefaultConfig();
...
```

Load a Default Global Store

```
public class MainApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        MobileSyncSDKManager.initNative(getApplicationContext(), MainActivity.class);
        MobileSyncSDKManager.getInstance().setupGlobalStoreFromDefaultConfig();
...
```

Hybrid

If SmartStore finds a soup configuration file, it automatically loads the file. To add the file to your project:

1. Copy the configuration file (`userstore.json` or `globalstore.json`) to the top-level `www/` directory of your hybrid project directory.
2. In a command prompt or Terminal window, change to your hybrid project directory and run: *`cordova prepare`*

Sample Code

MobileSyncExplorer and MobileSyncExplorerHybrid sample apps use a config file to set up SmartStore soups.

Note:

- Call the SmartStore loader method only if you are using a `userstore.json` to define soups. If you set up your soups with code instead of configuration files, don't call the loader method.
- Call the loader method after the customer has logged in.
- Do not call a loader method more than once.

Example: SmartStore uses the same configuration file—`userstore.json`—for native and hybrid versions of the MobileSyncExplorer sample. The final five paths in this configuration are required if you're using Mobile Sync.

```
{  "soups": [
    {
      "soupName": "contacts",
```

```
      "indexes": [
        { "path": "Id", "type": "string"},
        { "path": "FirstName", "type": "string"},
        { "path": "LastName", "type": "string"},
        { "path": "__local__", "type": "string"},
        { "path": "__locally_created__", "type": "string"},
        { "path": "__locally_updated__", "type": "string"},
        { "path": "__locally_deleted__", "type": "string"},
        { "path": "__sync_id__", "type": "integer"}
      ]
    }
  ]
}
```

IN THIS SECTION:

Registering a Soup through Code
Before you try to access a soup, you're required to register it.

SEE ALSO:

Preparing Soups for Mobile Sync

Registering a Soup through Code

Before you try to access a soup, you're required to register it.

To register a soup, you provide a soup name and a list of one or more index specifications. If, for some reason, you can't use a configuration file to define your soup structures, here's how you can do the same thing through code. Each of the following code examples builds an index spec array consisting of name, ID, and owner (or parent) ID fields.

You index a soup on one or more fields found in its entries. SmartStore makes sure that these indexes reflect any insert, update, and delete operations. Always specify at least one index field when registering a soup. For example, if you are using the soup as a simple key-value store, use a single index specification with a string type.

Note:

- As of Mobile SDK 6.0, you can register soups in native apps through a JSON configuration file. Where possible, we recommend this strategy over coding. See Registering Soups with Configuration Files.
- If your soup contains unusually large elements (> 1 MB), consider registering it to use external storage. See Using External Storage for Large Soup Elements.

Hybrid Apps

The JavaScript function for registering a soup requires callback functions for success and error conditions.

```
var sfSmartstore = function() {
    return cordova.require("com.salesforce.plugin.smartstore");
};
sfSmartstore().registerSoup(soupName, indexSpecs, successCallback, errorCallback);
```

If the soup does not exist, this function creates it. If the soup exists, registering lets you access it.

`indexSpecs`

Use the `indexSpecs` array to create the soup with predefined indexing. Entries in the `indexSpecs` array specify how to index the soup. Each entry consists of a `path:type` pair. `path` is the name of an index field; `type` is either "string", "integer", "floating", "full_text", or "json1".

```
var indexSpecs = [
    {path:"Name",type:"string"},
    {path:"Id",type:"string"}
];
```

Note:

- Index paths are case-sensitive and can include compound paths, such as `Owner.Name`.
- Index entries that are missing any fields described in an `indexSpecs` array are not tracked in that index.
- The type of the index applies only to the index. When you query an indexed field (for example, "`select {soup:path} from {soup}`"), the query returns data of the type that you specified in the index specification.
- Index columns can contain null fields.
- Beginning in Mobile SDK 4.1, you can specify index paths that point to *internal* (non-leaf) nodes. You can use these paths with `like` and `match` (full-text) queries. Use the `string` type when you define internal node paths.

  For example, consider this element in a soup named "spies":

  ```
  {
     "first_name":"James",
     "last_name":"Bond",
     "address":{
        "street_number":10,
        "street_name":"downing",
        "city":"london"
     }
  }
  ```

  In this case, "address" is an internal node because it has children. Through the index on the path "address", you can use a `like` or `match` query to find the "city" value—"london"—in "address". For example:

  ```
  SELECT {spies:first_name, spies:last_name} FROM spies WHERE {spies:address} LIKE
   'london'
  ```

- Beginning in Mobile SDK 4.1, you can include arrays in index paths, with some restrictions. See Using Arrays in Index Paths.

`successCallback`

The success callback function you supply takes one argument: the soup name. For example:

```
function(soupName) { alert("Soup " + soupName + " was successfully created"); }
```

When the soup is successfully created, `registerSoup()` calls the success callback function to indicate that the soup is ready. Wait to complete the transaction and receive the callback before you begin any activity. If you register a soup under the passed name, the success callback function returns the soup.

`errorCallback`

The error callback function takes one argument: the error description string.

```
function(err) { alert ("registerSoup failed with error: " + err); }
```

During soup creation, errors can happen for various reasons, including:

- An invalid or bad soup name
- No index (at least one index must be specified)
- Other unexpected errors, such as a database error

To find out if a soup exists, use:

```
navigator.smartstore.soupExists(soupName, successCallback, errorCallback);
```

Android Native Apps

For Android, you define index specs in an array of type `com.salesforce.androidsdk.smartstore.store.IndexSpec`. Each index spec comprises a path—the name of an index field—and a type. Index spec types are defined in the `SmartStore.Type` enum and include the following values:

- `string`
- `integer`
- `floating`
- `full_text`
- `json1`

```
public class OfflineStorage extends SalesforceActivity {
    private SmartStore smartStore;
    final IndexSpec[] ACCOUNTS_INDEX_SPEC = {
        new IndexSpec("Name", SmartStore.Type.string),
        new IndexSpec("Id", SmartStore.Type.string),
        new IndexSpec("OwnerId", SmartStore.Type.string)
    };

    public OfflineStorage() {
        smartStore = SmartStoreSDKManager.getInstance().getSmartStore();
        smartStore.registerSoup("Account", ACCOUNTS_INDEX_SPEC);
    }

    // ...
}
```

iOS Native Apps

For iOS, you define index specs in an array of `SFSoupIndex` objects. Each index spec comprises a path—the name of an index field—and a type. Index spec types are defined as constants in the `SFSoupIndex` class and include the following values:

- `kSoupIndexTypeString`
- `kSoupIndexTypeInteger`
- `kSoupIndexTypeFloating`
- `kSoupIndexTypeFullText`
- `kSoupIndexTypeJSON1`

Swift

In Mobile SDK 8.0 and later, a SmartStore native Swift extension provides the following soup registration method:

```
func createAccountsSoup(name: String) {
    guard let user = UserAccountManager.shared.currentUserAccount,
        let store = SmartStore.shared(withName: SmartStore.defaultStoreName,
            forUserAccount: user),
        let  index1 = SoupIndex(path: "Name", indexType: "String",
            columnName: "Name"),
        let  index2 = SoupIndex(path: "Id", indexType: "String",
            columnName: "Id")
    else {
        return
    }

    do {
        try store.registerSoup(withName: name, withIndices:[index1,index2])
    } catch (let error) {
        SalesforceLogger.d(RootViewController.self,
            message:"Couldn't create soup \(name).
                Error: \(error.localizedDescription)")
        return
    }
}
```

Objective-C

```
NSString* const kAccountSoupName = @"Account";

...
- (SFSmartStore *)store
{
    return [SFSmartStore sharedStoreWithName:kDefaultSmartStoreName];
}

...
- (void)createAccountsSoup {
    if (![self.store soupExists:kAccountSoupName]) {
        NSArray *keys = @[@"path", @"type"];
        NSArray *nameValues = @[@"Name", kSoupIndexTypeString];
        NSDictionary *nameDictionary = [NSDictionary
            dictionaryWithObjects:nameValues forKeys:keys];

        NSArray *idValues = @[@"Id", kSoupIndexTypeString];
        NSDictionary *idDictionary =
            [NSDictionary dictionaryWithObjects:idValues forKeys:keys];

        NSArray *ownerIdValues = @[@"OwnerId", kSoupIndexTypeString];
        NSDictionary *ownerIdDictionary =
            [NSDictionary dictionaryWithObjects:ownerIdValues
                forKeys:keys];

        NSArray *accountIndexSpecs =
```

```
            [SFSoupIndex asArraySoupIndexes:@[nameDictionary,
                idDictionary, ownerIdDictionary]];

        NSError* error = nil;
        [self.store registerSoup:kAccountSoupName
                  withIndexSpecs:accountIndexSpecs
                           error:&error];
        if (error) {
            NSLog(@"Cannot create SmartStore soup '%@'\nError: '%@'",
                kAccountSoupName, error.localizedDescription);
        }
    }
}
```

SEE ALSO:

SmartStore Data Types

Using Full-Text Search Queries

Using Arrays in Index Paths

Index paths can contain arrays, but certain rules apply.

Before Mobile SDK 4.1, index paths supported only maps—in other words, dictionaries or associative arrays. For example, in a path such as `a.b.c`, SmartStore required both `b` and `c` to be maps. Otherwise, when evaluating the path, SmartStore returned nothing.

In Mobile SDK 4.1 and later, index paths can contain arrays and maps. In the `a.b.c` example, if the value of `b` is an array, SmartStore expects the array to contain maps that define `c`. SmartStore then returns an array containing values of `c` keys found in the `b` array's maps.

Note: You can't use index paths that traverse arrays with JSON1 index specs.

Example: The following table shows various examples of `a.b.c` paths and the values returned by a SmartStore query.

Description	Example soup element	Value for path `a.b.c`
No arrays	`{ "a":{ "b":{ "c":1 } } }`	1
`c` points to an array (internal node).	`{ "a":{ "b":{ "c":[1,2,3] } } }`	`[ 1, 2, 3 ]`

Description	Example soup element	Value for path `a.b.c`

`b` points to an array of maps. Some maps contain the `c` key. Other maps are ignored.	<pre>{
	  "a":{
	  "b":[
	  {
	  "c":1
	  },
	  {
	  "c":2
	  },
	  {
	  "no-c":3
	  }
	  ]
	  }
	}</pre>
		<pre>[
		  1,
		  2
		]</pre>

`a` points to an array of maps. In some maps, `b` points to a map containing a key. In other maps, `b` points to an array of maps. Only values from `c` keys are returned.	<pre>{
	  "a":[
	  {
	  "b":{
	  "c":0
	  }
	  },
	  {
	  "b":[
	  {
	  "c":1
	  },
	  {
	  "c":2
	  },
	  {
	  "no-c":3
	  }
	  ]
	  }
	  ]
	}</pre>
		<pre>[
		  0,
		  [
		  1,
		  2
		  ]
		]</pre>

Populating a Soup

To add Salesforce records to a soup for offline access, use the REST API in conjunction with SmartStore APIs.

When you register a soup, you create an empty named structure in memory that's waiting for data. You typically initialize the soup with data from a Salesforce organization. To obtain the Salesforce data, you use Mobile SDK's standard REST request mechanism. When a successful REST response arrives, you extract the data from the response object and then upsert it into your soup.

Hybrid Apps

Hybrid apps use SmartStore functions defined in the `force.js` library. In this example, the click handler for the Fetch Contacts button calls `force.query()` to send a simple SOQL query ("`SELECT Name, Id FROM Contact`") to Salesforce. This call designates `onSuccessSfdcContacts(response)` as the callback function that receives the REST response. The `onSuccessSfdcContacts(response)` function iterates through the returned records in `response` and populates UI controls with Salesforce values. Finally, it upserts all records from the response into the sample soup.

```
force.query("SELECT Name,Id FROM Contact",
    onSuccessSfdcContacts, onErrorSfdc); var sfSmartstore = function() {
    return cordova.require("com.salesforce.plugin.smartstore");};
function onSuccessSfdcContacts(response) {
    logToConsole()("onSuccessSfdcContacts: received " +
        response.totalSize + " contacts");
    var entries = [];

    response.records.forEach(function(contact, i) {
           entries.push(contact);
    });

    if (entries.length > 0) {
        sfSmartstore().upsertSoupEntries(CONTACTS_SOUP_NAME,
            entries,
            function(items) {
                var statusTxt = "upserted: " + items.length +
                    " contacts";
                logToConsole()(statusTxt);
            },
         onErrorUpsert);
    }
}

function onErrorSfdc(param) {
    logToConsole()("onErrorSfdc: " + param);
}function onErrorUpsert(param) {
    logToConsole()("onErrorUpsert: " + param);
}
```

iOS Native Apps

iOS native apps use the `SFRestAPI` protocol for REST API interaction. The following code creates and sends a REST request for the SOQL query `SELECT Name, Id, OwnerId FROM Account`. If the request is successful, Salesforce sends the REST response to the `requestForQuery:send:delegate:` delegate method. The response is parsed, and each returned record is upserted into the SmartStore soup.

```
- (void)requestAccounts
{
    SFRestRequest *request = [[SFRestAPI sharedInstance]
        requestForQuery:@"SELECT Name, Id, OwnerId FROM Account"];
    [[SFRestAPI sharedInstance] send:request delegate:self];
}

//SFRestAPI protocol for successful response
```

```
- (void)request:(SFRestRequest *)request didLoadResponse:(id)dataResponse
{
    NSArray *records = dataResponse[@"records"];
    if (nil != records) {
        for (int i = 0; i < records.count; i++) {
            [self.store upsertEntries:@[records[i]]
                              toSoup:kAccountSoupName];
        }
    }
}
```

Android Native Apps

For REST API interaction, Android native apps typically use the `RestClient.sendAsync()` method with an anonymous inline definition of the `AsyncRequestCallback` interface. The following code creates and sends a REST request for the SOQL query `SELECT Name, Id, OwnerId FROM Account`. If the request is successful, Salesforce sends the REST response to the provided `AsyncRequestCallback.onSuccess()` callback method. The response is parsed, and each returned record is upserted into the SmartStore soup.

```
private void sendRequest(String soql, final String obj)
throws UnsupportedEncodingException {
    final RestRequest restRequest =
        RestRequest.getRequestForQuery(
            getString(R.string.api_version),
            "SELECT Name, Id, OwnerId FROM Account", "Account");
    client.sendAsync(restRequest, new AsyncRequestCallback() {
        @Override
        public void onSuccess(RestRequest request,
            RestResponse result) {
            // Consume before going back to main thread
            // Not required if you don't do main (UI) thread tasks here
            result.consumeQuietly();
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    // Network component doesn’t report app layer status.
                    // Use the Mobile SDK RestResponse.isSuccess() method to check
                    // whether the REST request itself succeeded.
                    if (result.isSuccess()) {
                        try {
                            final JSONArray records =
                                result.asJSONObject().getJSONArray("records");
                            insertAccounts(records);
                        } catch (Exception e) {
                            onError(e);
                        } finally {
                            Toast.makeText(MainActivity.this,
                                "Records ready for offline access.",
                                Toast.LENGTH_SHORT).show();
                        }
                    }
                }
            });
```

```
        }

        @Override
        public void onError(Exception e) {
            // You might want to log the error
            // or show it to the user
        }
    });
}

/**
 * Inserts accounts into the accounts soup.
 *
 * @param accounts Accounts.
 */
public void insertAccounts(JSONArray accounts) {
    try {
        if (accounts != null) {
            for (int i = 0; i < accounts.length(); i++) {
                if (accounts[i] != null) {
                    try {
                        smartStore.upsert(
                            ACCOUNTS_SOUP, accounts[i]);
                    } catch (JSONException exc) {
                        Log.e(TAG,
                            "Error occurred while attempting "
                            + "to insert account. Please verify "
                            + "validity of JSON data set.");
                    }
                }
            }
        }
    } catch (JSONException e) {
        Log.e(TAG, "Error occurred while attempting to "
            + "insert accounts. Please verify validity "
            + "of JSON data set.");
    }
}
```

Retrieving Data from a Soup

SmartStore provides a set of helper methods that build query strings for you.

For retrieving data from a soup, SmartStore provides helper functions that build query specs for you. A query spec is similar to an index spec, but contains more information about the type of query and its parameters. Query builder methods produce specs that let you query:

- Everything (”all” query)
- Using a Smart SQL
- For exact matches of a key (”exact” query)
- For full-text search on given paths (”match” query)
- For a range of values (”range” query)

- For wild-card matches ("like" query)

To query for a set of records, call the query spec factory method that suits your specifications. You can optionally define the index field, sort order, and other metadata to be used for filtering, as described in the following table:

Parameter	Description
`selectPaths` or `withSelectPaths`	(Optional in JavaScript) Narrows the query scope to only a list of fields that you specify. See Narrowing the Query to Return a Subset of Fields.
`indexPath` or `path`	Describes what you're searching for; for example, a name, account number, or date.
`beginKey`	(Optional in JavaScript) Used to define the start of a range query.
`endKey`	(Optional in JavaScript) Used to define the end of a range query.
`matchKey`	(Optional in JavaScript) Used to specify the search string in an exact or match query.
`orderPath`	(Optional in JavaScript—defaults to the value of the `path` parameter) For exact, range, and like queries, specifies the indexed path field to be used for sorting the result set. To query without sorting, set this parameter to a null value. Note: Mobile SDK versions 3.2 and earlier sort all queries on the indexed path field specified in the query.
`order`	(Optional in JavaScript) • JavaScript: Either "ascending" (default) or "descending." • iOS: Either `kSFSoupQuerySortOrderAscending` or `kSFSoupQuerySortOrderDescending`. • Android: Either `Order.ascending` or `Order.descending`.
`pageSize`	(Optional in JavaScript. If not present, the native plug-in calculates an optimal value for the resulting `Cursor.pageSize`) Number of records to return in each page of results.

For example, consider the following `buildRangeQuerySpec()` JavaScript call:

```
navigator.smartstore.buildRangeQuerySpec(
    "name", "Aardvark", "Zoroastrian", "ascending", 10, "name");
```

This call builds a range query spec that finds entries with names between Aardvark and Zoroastrian, sorted on the `name` field in ascending order:

```
{
   "querySpec":{
      "queryType":"range",
      "indexPath":"name",
      "beginKey":"Aardvark",
      "endKey":"Zoroastrian",
      "orderPath":"name",
      "order":"ascending",
      "pageSize":10
   }
}
```

In JavaScript `build*` functions, you can omit optional parameters only at the end of the function call. You can't skip one or more parameters and then specify the next without providing a dummy or null value for each option you skip. For example, you can use these calls:

- `buildAllQuerySpec(indexPath)`
- `buildAllQuerySpec(indexPath, order)`
- `buildAllQuerySpec(indexPath, order, pageSize)`
- `buildAllQuerySpec(indexPath, order, pageSize, selectPaths)`

However, you can't use this call because it omits the `order` parameter:

```
buildAllQuerySpec(indexPath, pageSize)
```

Note: All parameterized queries are single-predicate searches. Only Smart SQL queries support joins.

Query Everything

Traverses everything in the soup.

See Working with Query Results for information on page sizes.

Note: As a base rule, set `pageSize` to the number of entries you want displayed on the screen. For a smooth scrolling display, you can to increase the value to two or three times the number of entries shown.

JavaScript:

`buildAllQuerySpec(indexPath, order, pageSize, selectPaths)` returns all entries in the soup, with no particular order. `order` and `pageSize` are optional, and default to "ascending" and 10, respectively. The `selectPaths` argument is also optional.

iOS native:

```
+ (SFQuerySpec*) newAllQuerySpec:(NSString*)soupName
                        withPath:(NSString*)path
                       withOrder:(SFSoupQuerySortOrder)order
                   withPageSize:(NSUInteger)pageSize;

+ (SFQuerySpec*) newAllQuerySpec:(NSString*)soupName
                  withSelectPaths:(NSArray*)selectPaths
                    withOrderPath:(NSString*)orderPath
                        withOrder:(SFSoupQuerySortOrder)order
                     withPageSize:(NSUInteger)pageSize;
```

Android native:

```
public static QuerySpec buildAllQuerySpec(
    String soupName,
    String path,
    Order order,
    int pageSize)

public static QuerySpec buildAllQuerySpec(
    String soupName,
    String[] selectPaths,
    String orderPath,
```

```
    Order order,
    int pageSize);
```

Query with a Smart SQL SELECT Statement

Executes the query specified by the given Smart SQL statement. This function allows greater flexibility than other query factory functions because you provide your own SELECT statement. See Smart SQL Queries.

The following sample code shows a Smart SQL query that calls the SQL `COUNT` function.

JavaScript:

```
var querySpec =
    navigator.smartstore.buildSmartQuerySpec(
        "select count(*) from {employees}", 1);

navigator.smartstore.runSmartQuery(querySpec, function(cursor) {
    // result should be [[ n ]] if there are n employees
});
```

In JavaScript, `pageSize` is optional and defaults to 10.

iOS native:

In Mobile SDK 8.0 and later, the native Swift SmartStore extension provides two ways to run a Smart SQL query.

Swift

In iOS 12.2 or later:

```
public func query(_ smartSql: String) -> Result<[Any], SmartStoreError>
```

In iOS 13.0 or later, using Combine Publisher:

```
public func publisher(for smartSql: String) -> Future<[Any], SmartStoreError>
```

Objective-C

```
SFQuerySpec* querySpec =
    [SFQuerySpec
        newSmartQuerySpec:@"select count(*) from {employees}"
             withPageSize:1];
NSArray* result = [_store queryWithQuerySpec:querySpec pageIndex:0 error:nil];
// result should be [[ n ]] if there are n employees
```

Android native:

```
try {
    JSONArray result =
        store.query(QuerySpec.buildSmartQuerySpec(
            "select count(*) from {Accounts}", 1), 0);
    // result should be [[ n ]] if there are n employees
    Log.println(Log.INFO, "REST Success!", "\nFound " +
        result.getString(0) + " accounts.");
} catch (JSONException e) {
    Log.e(TAG, "Error occurred while counting the number of account records. "
        +  "Please verify validity of JSON data set.");
}
```

Query by Exact

Finds entries that exactly match the given `matchKey` for the `indexPath` value. You use this method to find child entities of a given ID. For example, you can find opportunities by `Status`.

JavaScript:

In JavaScript, you can set the `order` parameter to either "ascending" or "descending". `order`, `pageSize`, and `orderPath` are optional, and default to "ascending", 10, and the `path` argument, respectively. The `selectPaths` argument is also optional.

```
navigator.smartstore.buildExactQuerySpec(
    path, matchKey, pageSize, order, orderPath, selectPaths)
```

The following JavaScript code retrieves children by ID:

```
var querySpec = navigator.smartstore.buildExactQuerySpec(
   "sfdcId",
   "some-sfdc-id");
navigator.smartstore.querySoup("Catalogs",
   querySpec, function(cursor) {
   // we expect the catalog to be in:
   // cursor.currentPageOrderedEntries[0]
});
```

The following JavaScript code retrieves children by parent ID:

```
var querySpec = navigator.smartstore.buildExactQuerySpec("parentSfdcId", "some-sfdc-id);
navigator.smartstore.querySoup("Catalogs", querySpec, function(cursor) {});
```

iOS native:

In iOS, you can set the `order` parameter to either `kSFSoupQuerySortOrderAscending` or `kSFSoupQuerySortOrderDescending`. To narrow the query's scope to certain fields, use the second form and pass an array of field names through the `withSelectPaths` parameter.

```
+ (SFQuerySpec*) newExactQuerySpec:(NSString*)soupName
                          withPath:(NSString*)path
                      withMatchKey:(NSString*)matchKey
                     withOrderPath:(NSString*)orderPath
                         withOrder:(SFSoupQuerySortOrder)order
                      withPageSize:(NSUInteger)pageSize;

+ (SFQuerySpec*) newExactQuerySpec:(NSString*)soupName
                    withSelectPaths:(NSArray*)selectPaths
                           withPath:(NSString*)path
                       withMatchKey:(NSString*)matchKey
                      withOrderPath:(NSString*)orderPath
                          withOrder:(SFSoupQuerySortOrder)order
                       withPageSize:(NSUInteger)pageSize;
```

Android native:

In Android, you can set the `order` parameter to either `Order.ascending` or `Order.descending`. To narrow the query's scope to certain fields, use the second form and pass an array of field names through the `selectPaths` parameter.

```
public static QuerySpec buildExactQuerySpec(
    String soupName, String path, String exactMatchKey,
    String orderPath, Order order, int pageSize)
```

```
public static QuerySpec buildExactQuerySpec(
    String soupName, String[] selectPaths, String path,
    String exactMatchKey, String orderPath,
    Order order, int pageSize);
```

Query by Match

Finds entries that exactly match the full-text search query in `matchKey` for the `indexPath` value. See Using Full-Text Search Queries.

JavaScript:

In JavaScript, you can set the `order` parameter to either "ascending" or "descending". `order`, `pageSize`, and `orderPath` are optional, and default to "ascending", 10, and the `path` argument, respectively. The `selectPaths` argument is also optional.

```
navigator.smartstore.buildMatchQuerySpec(
    path, matchKey, order, pageSize, orderPath, selectPaths)
```

iOS native:

In iOS, you can set the `order` parameter to either `kSFSoupQuerySortOrderAscending` or `kSFSoupQuerySortOrderDescending`. To narrow the query's scope to certain fields, use the second form and pass an array of field names through the `withSelectPaths` parameter.

```
+ (SFQuerySpec*) newMatchQuerySpec:(NSString*)soupName
                          withPath:(NSString*)path
                      withMatchKey:(NSString*)matchKey
                     withOrderPath:(NSString*)orderPath
                         withOrder:(SFSoupQuerySortOrder)order
                      withPageSize:(NSUInteger)pageSize;

+ (SFQuerySpec*) newMatchQuerySpec:(NSString*)soupName
                    withSelectPaths:(NSArray*)selectPaths
                           withPath:(NSString*)path
                       withMatchKey:(NSString*)matchKey
                      withOrderPath:(NSString*)orderPath
                          withOrder:(SFSoupQuerySortOrder)order
                       withPageSize:(NSUInteger)pageSize;
```

Android native:

In Android, you can set the `order` parameter to either `Order.ascending` or `Order.descending`. To narrow the query's scope to certain fields, use the second form and pass an array of field names through the `selectPaths` parameter.

```
public static QuerySpec buildMatchQuerySpec(
    String soupName, String path, String exactMatchKey,
    String orderPath, Order order, int pageSize)

public static QuerySpec buildMatchQuerySpec(
    String soupName, String[] selectPaths, String path,
    String matchKey, String orderPath, Order order,
    int pageSize)
```

Query by Range

Finds entries whose `indexPath` values fall into the range defined by `beginKey` and `endKey`. Use this function to search by numeric ranges, such as a range of dates stored as integers.

By passing null values to `beginKey` and `endKey`, you can perform open-ended searches:

- To find all records where the field at `indexPath` is greater than or equal to `beginKey`, pass a null value to `endKey`.
- To find all records where the field at `indexPath` is less than or equal to `endKey`, pass a null value to `beginKey`.
- To query everything, pass a null value to both `beginKey` and `endKey`.

JavaScript:

In JavaScript, you can set the `order` parameter to either "ascending" or "descending". `order`, `pageSize`, and `orderPath` are optional, and default to "ascending", 10, and the `path` argument, respectively. The `selectPaths` argument is also optional.

```
navigator.smartstore.buildRangeQuerySpec(
    path, beginKey, endKey, order, pageSize, orderPath, selectPaths)
```

iOS native:

In iOS, you can set the `order` parameter to either `kSFSoupQuerySortOrderAscending` or `kSFSoupQuerySortOrderDescending`. To narrow the query's scope to certain fields, use the second form and pass an array of field names through the `withSelectPaths` parameter.

```
+ (SFQuerySpec*) newRangeQuerySpec:(NSString*)soupName
                          withPath:(NSString*)path
                     withBeginKey:(NSString*)beginKey
                       withEndKey:(NSString*)endKey
                    withOrderPath:(NSString*)orderPath
                        withOrder:(SFSoupQuerySortOrder)order
                     withPageSize:(NSUInteger)pageSize;

+ (SFQuerySpec*) newRangeQuerySpec:(NSString*)soupName
                  withSelectPaths:(NSArray*)selectPaths
                          withPath:(NSString*)path
                     withBeginKey:(NSString*)beginKey
                       withEndKey:(NSString*)endKey
                    withOrderPath:(NSString*)orderPath
                        withOrder:(SFSoupQuerySortOrder)order
                     withPageSize:(NSUInteger)pageSize;
```

Android native:

In Android, you can set the `order` parameter to either `Order.ascending` or `Order.descending`. To narrow the query's scope to certain fields, use the second form and pass an array of field names through the `selectPaths` parameter.

```
public static QuerySpec buildRangeQuerySpec(
    String soupName, String path, String beginKey,
    String endKey, String orderPath, Order order, int pageSize)

public static QuerySpec buildRangeQuerySpec(
    String soupName, String[] selectPaths, String path,
    String beginKey, String endKey, String orderPath,
    Order order, int pageSize);
```

Query by Like

Finds entries whose `indexPath` values are like the given `likeKey`. You can use the "%" wild card to search for partial matches as shown in these syntax examples:

- To search for terms that begin with your keyword: "foo%"
- To search for terms that end with your keyword: "%foo"
- To search for your keyword anywhere in the `indexPath` value: "%foo%"

. Use this function for general searching and partial name matches. Use the query by "match" method for full-text queries and fast queries over large data sets.

Note: Query by "like" is the slowest query method.

JavaScript:

In JavaScript, you can set the `order` parameter to either "ascending" or "descending". `order`, `pageSize`, and `orderPath` are optional, and default to "ascending", 10, and the `path` argument, respectively. The `selectPaths` argument is also optional.

```
navigator.smartstore.buildLikeQuerySpec(
    path, likeKey, order, pageSize, orderPath, selectPaths)
```

iOS native:

In iOS, you can set the `order` parameter to either `kSFSoupQuerySortOrderAscending` or `kSFSoupQuerySortOrderDescending`. To narrow the query's scope to certain fields, use the second form and pass an array of field names through the `withSelectPaths` parameter.

```
+ (SFQuerySpec*) newLikeQuerySpec:(NSString*)soupName
                         withPath:(NSString*)path
                      withLikeKey:(NSString*)likeKey
                    withOrderPath:(NSString*)orderPath
                        withOrder:(SFSoupQuerySortOrder)order
                     withPageSize:(NSUInteger)pageSize;

+ (SFQuerySpec*) newLikeQuerySpec:(NSString*)soupName
                   withSelectPaths:(NSArray*)selectPaths
                         withPath:(NSString*)path
                      withLikeKey:(NSString*)likeKey
                    withOrderPath:(NSString*)orderPath
                        withOrder:(SFSoupQuerySortOrder)order
                     withPageSize:(NSUInteger)pageSize;
```

Android native:

In Android, you can set the `order` parameter to either `Order.ascending` or `Order.descending`. To narrow the query's scope to certain fields, use the second form and pass an array of field names through the `selectPaths` parameter.

```
public static QuerySpec buildLikeQuerySpec(
    String soupName, String path, String likeKey,
    String orderPath, Order order, int pageSize)

public static QuerySpec buildLikeQuerySpec(
    String soupName, String[] selectPaths,
    String path, String likeKey, String orderPath,
    Order order, int pageSize)
```

Executing the Query

In JavaScript, queries run asynchronously and return a cursor to your success callback function, or an error to your error callback function. The success callback takes the form `function(cursor)`. You use the `querySpec` parameter to pass your query specification to the `querySoup` method.

```
navigator.smartstore.querySoup(soupName, querySpec,
    successCallback, errorCallback);
```

Narrowing the Query to Return a Subset of Fields

In Smart SQL query specs, you can limit the list of fields that the query returns by specifying the fields in the Smart SQL statement. For other types of query specs, you can do the same thing with the `selectPaths` parameter. When this argument is used, the method returns an array of arrays that contains an array for each element that satisfies the query. Each element array includes only the fields specified in `selectPaths`. This parameter is available for "all", "exact", "match", "range", and "like" query specs.

Here's an example. Consider a soup that contains elements such as the following:

```
{"_soupEntryId":1, "name":"abc", "status":"active", ...},
{"_soupEntryId":2, "name":"abd", "status":"inactive", ...}, ...
```

Let's run a "like" query that uses "ab%" as the LIKE key and `name` as the path. This query returns an array of objects, each of which contains an entire element:

```
[ {"_soupEntryId":1, "name": "abc", "status":"active",...},
{"_soupEntryId":2, "name":"abd", "status":"inactive",...},
...]
```

Now let's refine the query by adding `_soupEntryId` and `name` as selected paths. The query now returns a more efficient array of arrays with only the `_soupEntryId` and `name` field values:

```
[[1, "abc"], [2, "abd"], ...]
```

Retrieving Individual Soup Entries by Primary Key

All soup entries are automatically given a unique internal ID (the primary key in the internal table that holds all entries in the soup). That ID field is made available as the `_soupEntryId` field in the soup entry.

To look up soup entries by `_soupEntryId` in JavaScript, use the `retrieveSoupEntries` function. This function provides the fastest way to retrieve a soup entry, but it's usable only when you know the `_soupEntryId`:

```
navigator.smartStore.retrieveSoupEntries(soupName, indexSpecs,
    successCallback, errorCallback)
```

The return order is not guaranteed. Also, entries that have been deleted are not returned in the resulting array.

Smart SQL Queries

To exert full control over your queries—or to reuse existing SQL queries—you can define custom SmartStore queries.

SmartStore supports a Smart SQL query language for free-form SELECT statements. Smart SQL queries combine standard SQL SELECT grammar with additional descriptors for referencing soups and soup fields. This approach gives you maximum control and flexibility, including the ability to use joins. Smart SQL supports all standard SQL SELECT constructs.

As of Mobile SDK 9.1, Smart SQL no longer requires index paths for all fields referenced in SELECT or WHERE clauses, except as noted in the following restrictions.

Smart SQL Restrictions

- For soups that use the deprecated external storage feature, Smart SQL still requires index paths for any fields referenced in SELECT or WHERE clauses.

  Warning: External storage is deprecated in Mobile SDK 10.0 and will be removed in Mobile SDK 11.0. See Using External Storage for Large Soup Elements.

- You can't write MATCH queries with Smart SQL. For example, the following query doesn't work: `SELECT {soupName:_soup} FROM {soupName} WHERE {soupName:name} MATCH 'cat'`

Syntax

Syntax is identical to the standard SQL SELECT specification but with the following adaptations:

Usage	Syntax
To specify a column	`{<soupName>:<path>}`
To specify a table	`{<soupName>}`
To refer to the entire soup entry JSON string	`{<soupName>:_soup}`
To refer to the internal soup entry ID	`{<soupName>:_soupEntryId}`
To refer to the last modified date	`{<soupName>:_soupLastModifiedDate}`

Sample Queries

Consider two soups: one named Employees, and another named Departments. The Employees soup contains standard fields such as:

- First name (`firstName`)
- Last name (`lastName`)
- Department code (`deptCode`)
- Employee ID (`employeeId`)
- Manager ID (`managerId`)

The Departments soup contains:

- Name (`name`)
- Department code (`deptCode`)

Here are some examples of basic Smart SQL queries using these soups:

```
select {employees:firstName}, {employees:lastName}
from {employees} order by {employees:lastName}

select {departments:name}
from {departments}
order by {departments:deptCode}
```

Joins

Smart SQL also allows you to use joins. For example:

```
select {departments:name}, {employees:firstName} || ' ' || {employees:lastName}
from {employees}, {departments}
where {departments:deptCode} = {employees:deptCode}
order by {departments:name}, {employees:lastName}
```

You can even do self-joins:

```
select mgr.{employees:lastName}, e.{employees:lastName}
from {employees} as mgr, {employees} as e
where mgr.{employees:employeeId} = e.{employees:managerId}
```

Note: Doing a join on a JSON1 index requires a slightly extended syntax. For example, instead of

```
select {soup1:path1} from {soup1}, {soup2}
```

use

```
select {soup1}.{soup1:path1} from {soup1}, {soup2}
```

Aggregate Functions

Smart SQL supports the use of aggregate functions such as:

- COUNT
- SUM
- AVG

For example:

```
select {account:name},
    count({opportunity:name}),
    sum({opportunity:amount}),
    avg({opportunity:amount}),
    {account:id},
    {opportunity:accountid}
from {account},
    {opportunity}
where {account:id} = {opportunity:accountid}
group by {account:name}
```

Using Full-Text Search Queries

To perform efficient and flexible searches in SmartStore, you use full-text queries. Full-text queries yield significant performance advantages over "like" queries when you're dealing with large data sets.

Beginning with Mobile SDK 3.3, SmartStore supports full-text search. Full-text search is a technology that internet search engines use to collate documents placed on the web.

About Full-Text

Here's how full-text search works: A customer inputs a term or series of terms. Optionally, the customer can connect terms with binary operators or group them into phrases. A full-text search engine evaluates the given terms, applying any specified operators and groupings. The search engine uses the resulting search parameters to find matching documents, or, in the case of SmartStore, matching soup elements. To support full text search, SmartStore provides a full-text index spec for defining soup fields, and a query spec for performing queries on those fields.

Full-text queries, or "match" queries, are more efficient than "like" queries. "Like" queries require full index scans of all keys, with run times proportional to the number of rows searched. "Match" queries find the given term or terms in the index and return the associated record IDs. The full-text search optimization is negligible for fewer than 1000 records, but, beyond that threshold, run time stays nearly constant as the number of records increases. If you're searching through tens of thousands of records, “match” queries can be 10–100 times faster than “like” queries.

Keep these points in mind when using full-text fields and queries:

- Insertions with a full-text index field take longer than ordinary insertions.
- You can't perform MATCH queries in a Smart SQL statement. For example, the following query is **not supported**:

  ```
  SELECT {soupName:_soup} FROM {soupName} WHERE {soupName:name} MATCH 'cat'
  ```

  Instead, use a “match” query spec.

Staying Current with Full-Text Search

In Mobile SDK 4.2, SmartStore updates its full-text search version from FTS4 to FTS5. This upgrade lets Mobile SDK take advantage of full-text index specs.

If you upgrade an app from Mobile SDK 4.1 to 4.2, existing FTS4 virtual tables remain intact. On the other hand, new soups that you create after upgrading use FTS5 virtual tables. These soups all work seamlessly together, but you can choose to upgrade legacy soups. Simply call `alterSoup` and pass in your original set of index specs. This call uses FTS5 to recreate the virtual tables that back full-text index specs.

See “Appendix A” at www.sqlite.org/fts5.html for a comparison of FTS4 to FTS5.

IN THIS SECTION:

Full-Text Search Index Specs

To use full-text search, you register your soup with one or more full-text-indexed paths. SmartStore provides a *full_text* index spec for designating index fields.

When you define a path with a full-text index, you can also use that path for non-full-text queries. These other types of queries—”all”, “exact”, ":like", “range”, and “smart” queries—interpret full-text indexed fields as string indexed fields.

The following examples show how to instantiate a full-text index spec.

Example: **iOS:**

```
[[SFSoupIndex alloc]
    initWithDictionary:@{kSoupIndexPath: @"some_path",
    kSoupIndexType: kSoupIndexTypeFullText}]
```

Android:

```
new IndexSpec("some_path", Type.full_text)
```

JavaScript:

```
new navigator.smartstore.SoupIndexSpec("some_path", "full_text")
```

Full-Text Query Specs

To perform a full-text query, you create a SmartStore "match" query spec using your platform's match query method. For the `matchKey` argument, you provide a full-text search query.

Use the following methods to create full-text query specs.

iOS:

```
+ (SFQuerySpec*) newMatchQuerySpec:(NSString*)soupName
                          withPath:(NSString*)path
                      withMatchKey:(NSString*)matchKey
                     withOrderPath:(NSString*)orderPath
                         withOrder:(SFSoupQuerySortOrder)order
                      withPageSize:(NSUInteger)pageSize;

+ (SFQuerySpec*) newMatchQuerySpec:(NSString*)soupName
                   withSelectPaths:(NSArray*)selectPaths
                          withPath:(NSString*)path
                      withMatchKey:(NSString*)matchKey
                     withOrderPath:(NSString*)orderPath
                         withOrder:(SFSoupQuerySortOrder)order
                      withPageSize:(NSUInteger)pageSize;
```

Android:

```
public static QuerySpec buildMatchQuerySpec(
    String soupName, String path, String exactMatchKey,
    String orderPath, Order order, int pageSize)

public static QuerySpec buildMatchQuerySpec(
    String soupName, String[] selectPaths, String path,
    String matchKey, String orderPath, Order order,
    int pageSize)
```

JavaScript:

```
smartstore.buildMatchQuerySpec(
    path, matchKey, order, pageSize, orderPath, selectPaths)
```

Full-Text Query Syntax

Mobile SDK full-text queries use SQLite's enhanced query syntax. With this syntax, you can use logical operators to refine your search.

The following table shows the syntactical options that Mobile SDK queries support. Following the table are keyed examples of the various query styles and sample output. For more information, see Sections 3.1, "Full-text Index Queries," and 3.2, "Set Operations Using The Enhanced Query Syntax," at sqlite.org.

Query Option	SmartStore Behavior	Related Examples
Specify one or more full-text indexed paths	Performs match against values only at the paths you defined.	g, h, i, j, and k
Set the path to a null value	Performs match against all full-text indexed paths Note: If your path is null, you can still specify a target field in the `matchKey` argument. Use this format: `{soupName:path}:term`	a,b,c,d,e, and f
Specify more than one term without operators or grouping	Assumes an "AND" between terms	b and h
Place a star at the end of a term	Matches rows containing words that start with the query term	d and j
Use "OR" between terms	Finds one or both terms	c and i
Use the unary "NOT" operator before a term	Ignores rows that contain that term	e, f, and k
Specify a phrase search by placing multiple terms within double quotes (" ").	Returns soup elements in which the entire phrase occurs in one or more full-text indexed fields	

Example: For these examples, a soup named "animals" contains the following records. The `name` and `color` fields are indexed as `full_text`.

```
{"id": 1, "name": "cat", "color": "black"}
{"id": 2, "name": "cat", "color": "white"}
{"id": 3, "name": "dog", "color": "black"}
{"id": 4, "name": "dog", "color": "brown"}
{"id": 5, "name": "dog", "color": "white"}
```

Table 4: Query Syntax Examples

Example	Path	Match Key	Selects...	Records Returned
a.	null	"black"	Records containing the word "black" in any full-text indexed field	1, 3

Example	Path	Match Key	Selects...	Records Returned
b.	null	"black cat"	Records containing the words "black" and "cat" in any full-text indexed field	1
c.	null	"black OR cat"	Records containing either the word "black" or the word "cat" in any full-text indexed field	1, 2, 3
d.	null	"b*"	Records containing a word starting with "b" in any full-text indexed field	1, 3
e.	null	"black NOT cat"	Records containing the word "black" but not the word "cat" in any full-text indexed field	3
f.	null	"{animals:color}:black NOT cat"	Records containing the word "black" in the color field and not having the word "cat" in any full-text indexed field	3
g.	"color"	"black"	Records containing the word "black" in the `color` field	1, 3
h.	"color"	"black cat"	Records containing the words "black" and "cat" in the `color` field	No records
i.	"color"	"black OR cat"	Records containing either the word "black" or the word "cat" in the `color` field	1, 3
j.	"color"	"b*"	Records containing a word starting with "b" in the `color` field	1, 3
k.	"color"	"black NOT cat"	Records containing the word "black" but not the word "cat" in the `color` field	1, 3

Working with Query Results

Mobile SDK provides mechanisms on each platform that let you access query results efficiently, flexibly, and dynamically.

Often, a query returns a result set that is too large to load all at once into memory. In this case, Mobile SDK initially returns a small subset of the results—a single *page*, based on a size that you specify. You can then retrieve more pages of results and navigate forwards and backwards through the result set.

JavaScript:

When you perform a query in JavaScript, SmartStore returns a cursor object that lets you page through the query results. Your code can move forward and backwards through the cursor's pages. To navigate through cursor pages, use the following functions.

- `navigator.smartstore.moveCursorToPageIndex(cursor, newPageIndex, successCallback, errorCallback)` —Move the cursor to the page index given, where 0 is the first page, and `totalPages - 1` is the last page.
- `navigator.smartstore.moveCursorToNextPage(cursor, successCallback, errorCallback)` —Move to the next entry page if such a page exists.
- `navigator.smartstore.moveCursorToPreviousPage(cursor, successCallback, errorCallback)` —Move to the previous entry page if such a page exists.
- `navigator.smartstore.closeCursor(cursor, successCallback, errorCallback)` —Close the cursor when you're finished with it.

Note:

- The `successCallback` function accepts one argument: the updated cursor.
- Cursors are not static snapshots of data—they are dynamic. The only data the cursor holds is the original query and your current position in the result set. When you move your cursor, the query runs again. If you change the soup while paging through the cursor, the cursor shows those changes. You can even access newly created soup entries, assuming they satisfy the original query.

iOS native:

Internally, iOS native apps use the third-party `FMResultSet` class to obtain query results. When you call a SmartStore query spec method, use the `pageSize` parameter to control the amount of data that you get back from each call. To traverse pages of results, iteratively call the `queryWithQuerySpec:pageIndex:withDB:` or `queryWithQuerySpec:pageIndex:error:` method of the `SFSmartStore` class with the same query spec object while incrementing or decrementing the zero-based `pageIndex` argument.

Android native:

Internally, Android native apps use the `android.database.Cursor` interface for cursor manipulations. When you call a SmartStore query spec method, use the `pageSize` parameter to control the amount of data that you get back from each call. To traverse pages of results, iteratively call the `SmartStore.query()` method with the same query spec object while incrementing or decrementing the zero-based `pageIndex` argument.

Inserting, Updating, and Upserting Data

SmartStore defines standard fields that help you track entries and synchronize soups with external servers.

System Fields: _soupEntryId and _soupLastModifiedDate

To track soup entries for insert, update, and delete actions, SmartStore adds a few fields to each entry:

- `_soupEntryId`—This field is the primary key for the soup entry in the table for a given soup.
- `_soupLastModifiedDate`, `_soupCreatedDate`—The number of milliseconds since 1/1/1970.
  - To convert a date value to a JavaScript date, use `new Date(entry._soupLastModifiedDate)`.
  - To convert a date to the corresponding number of milliseconds since 1/1/1970, use `date.getTime()`.

When you insert or update soup entries, SmartStore automatically sets these fields. When you remove or retrieve specific entries, you can reference them by `_soupEntryId`.

Beginning with Mobile SDK 4.2, SmartStore creates indexes on the `_soupLastModifiedDate` and `_soupCreatedDate` fields. These indexes provide a performance boost for queries that use these fields. In older soups, the `_soupLastModifiedDate` and `_soupCreatedDate` fields exist but are not indexed. To create these indexes to legacy soups, simply call `alterSoup` and pass in your original set of index specs.

About Upserting

To insert or update soup entries—letting SmartStore determine which action is appropriate—you use an *upsert* method.

If `_soupEntryId` is already set in any of the entries presented for upsert, SmartStore updates the soup entry that matches that ID. If an upsert entry doesn't have a `_soupEntryId` slot, or its `_soupEntryId` doesn't match an existing soup entry, SmartStore inserts the entry and overwrites its `_soupEntryId`.

Upserting with an External ID

If your soup entries mirror data from an external system, you usually refer to those entries by their external primary key IDs. For that purpose, SmartStore supports upsert with an external ID. When you perform an upsert, you can designate any index field as the external ID field. SmartStore looks for existing soup entries with the same value in the designated field with the following results:

- If no field with the same value is found, SmartStore creates a soup entry.
- If the external ID field is found, SmartStore updates the entry with the matching external ID value.
- If more than one field matches the external ID, SmartStore returns an error.

To create an entry locally, set the external ID field to a value that you can query when uploading the new entries to the server.

When you update the soup with external data, always use the external ID. Doing so guarantees that you don't end up with duplicate soup entries for the same remote record.

SmartStore also lets you track inter-object relationships. For example, imagine that you create a product offline that belongs to a catalog that doesn't yet exist on the server. You can capture the product's relationship with the catalog entry through the `parentSoupEntryId` field. Once the catalog exists on the server, you can capture the external relationship by updating the local product record's `parentExternalId` field.

Upsert Methods

JavaScript:

The `cordova.force.js` library provides two JavaScript upsert functions, each with one overload:

```
navigator.smartStore.upsertSoupEntries(isGlobalStore, soupName,
    entries[], successCallback, errorCallback)
navigator.smartStore.upsertSoupEntries(storeConfig, soupName,
    entries[], successCallback, errorCallback)
```

```
navigator.smartStore.upsertSoupEntriesWithExternalId(isGlobalStore, soupName,
entries[], externalPathId, successCallback, errorCallback)
navigator.smartStore.upsertSoupEntriesWithExternalId(storeConfig, soupName,
entries[], externalPathId, successCallback, errorCallback)
```

To upsert local data only, use the first `upsert()` function. To upsert data from an external server, use the second function, which supports the `externalPathId` parameter.

iOS native:

The iOS `SFSmartStore` class provides two instance methods for upserting. The first lets you specify all available options:

- Soup name
- `NSArray` object containing index specs
- Path for an external ID field name
- An output `NSError` object to communicate errors back to the app

Objective-C:

```
- (NSArray *)upsertEntries:(NSArray *)entries
                    toSoup:(NSString *)soupName
         withExternalIdPath:(NSString *)externalIdPath
                     error:(NSError **)error;
```

Swift:

```
func upsert(entries: [Any], forSoupNamed: String,
        withExternalIdPath: String) throws -> [Any]
```

Example:

```
var entries = store.upsert(entries: entries,forSoupNamed: soupName, withExternalIdPath:
path)
```

The second method uses the `_soupEntryId` field for the external ID path:

Objective-C:

```
- (NSArray *)upsertEntries:(NSArray *)entries
                    toSoup:(NSString *)soupName;
```

Swift:

```
func upsert(entries: [[AnyHashable : Any]], forSoupNamed: String) -> [[AnyHashable : Any]]
```

Example:

```
var entries = store.upsert(entries: entries,forSoupNamed: soupName)
```

Android native:

Android provides three overloads of its `upsert()` method. The first overload lets you specify all available options:

- Soup name
- JSON object containing one or more entries for upserting
- Path for an arbitrary external ID field name
- Flag indicating whether to use a transactional model for inserts and updates

```
public JSONObject upsert(
    String soupName, JSONObject soupElt, String externalIdPath,
        boolean handleTx)
    throws JSONException
```

The second overload enforces the use of a transactional model for inserts and updates:

```
public JSONObject upsert(
    String soupName, JSONObject soupElt, String externalIdPath)
    throws JSONException
```

The third overload enforces the transactional model and uses the `_soupEntryId` field for the external ID path:

```
public JSONObject upsert(
    String soupName, JSONObject soupElt)
    throws JSONException
```

Example: The following JavaScript code contains sample scenarios. First, it calls `upsertSoupEntries` to create an account soup entry. In the success callback, the code retrieves the new record with its newly assigned soup entry ID. It then changes the account description and calls `forcetk.mobilesdk` methods to create the account on the server and then update it. The final call demonstrates an upsert with external ID. To make the code more readable, no error callbacks are specified. Also, because all SmartStore calls are asynchronous, real applications perform each step in the success callback of the previous step.

This code uses the value `new` for the `id` field because the record doesn't yet exist on the server. When the app comes online, it can query for records that exist only locally (by looking for records where `id == "new"`) and upload them to the server. Once the server returns IDs for the new records, the app can update their `id` fields in the soup.

```
var sfSmartstore =
    function() {return cordova.require("com.salesforce.plugin.smartstore");};
// ...
// Specify data for the account to be created
var acc = {id: "new", Name: "Cloud Inc",
    Description: "Getting started"};

// Create account in SmartStore
// This upsert does a "create" because
// the account has no _soupEntryId field
sfSmartstore().upsertSoupEntries("accounts", [ acc ],
   function(accounts) {
      acc = accounts[0];
      // acc should now have a _soupEntryId field
      // (and a _lastModifiedDate as well)
});

// Update account's description in memory
acc["Description"] = "Just shipped our first app ";

// Update account in SmartStore
// This does an "update" because acc has a _soupEntryId field
sfSmartstore().upsertSoupEntries("accounts", [ acc ],
   function(accounts) {
      acc = accounts[0];
});

// Create account on server
// (sync client -> server for entities created locally)
force.create("account", {
   "Name": acc["Name"],
   "Description": acc["Description"]},
   function(result) {
      acc["id"] = result["id"];
      // Update account in SmartStore
      sfSmartstore().upsertSoupEntries("accounts", [ acc ]);
});

// Update account's description in memory
```

```
acc["Description"] = "Now shipping for iOS and Android";

// Update account's description on server
// Sync client -> server for entities existing on server
force.update("account", acc["id"],
   {"Description": acc["Description"]});

// Later, there is an account (with id: someSfdcId) that you want
// to get locally

// There might be an older version of that account in the
// SmartStore already

// Update account on client
// sync server -> client for entities that might or might not
// exist on client
force.retrieve(
   "account", someSfdcId, "id,Name,Description",
   function(result) {
      // Create or update account in SmartStore
      // (looking for an account with the same sfdcId)
      sfSmartstore().upsertSoupEntriesWithExternalId(
         "accounts", [result], "id");
});
```

Using External Storage for Large Soup Elements

Some years ago, Mobile SDK implemented external storage to address limitations with storing large JSON strings in the database—for example, a 1-MB cursor window limitation on Android. These limitations no longer exist. Furthermore, recent performance analysis shows that external storage is now *slower* than internal storage.

Mobile SDK no longer recommends using external storage with SmartStore soups. The external storage feature has been removed in Mobile SDK 11.0.

IN THIS SECTION:

Soup Specs

The external storage feature and soup specs have been removed in Mobile SDK 11.0.

Register a Soup with External Storage

The external storage feature has been removed in Mobile SDK 11.0.

Alter a Soup with External Storage

The external storage feature and `alterSoup` methods have been removed in Mobile SDK 11.0.

Removing Soup Elements

Traditionally, SmartStore methods let you remove soup elements by specifying an array of element IDs. To do so, you usually run a preliminary query to retrieve the candidate IDs, then call the method that performs the deletion. In Mobile SDK 4.2, SmartStore ups the game by adding a query option to its element deletion methods. With this option, you provide only a query, and SmartStore deletes all elements that satisfy that query. This approach delivers a performance boost because both the query and the deletion operation occur in a single call.

Hybrid Apps

In hybrid apps, you use the third parameter to pass either an ID array or a SmartStore query spec.

```
removeFromSoup([isGlobalStore, ]soupName, entryIdsOrQuerySpec,
    successCB, errorCB)
removeFromSoup([storeConfig, ]soupName, entryIdsOrQuerySpec,
    successCB, errorCB)
```

In addition to success and error callbacks, this function takes the following arguments:

Table 5: Parameters

Parameter Name	Argument Description
`isGlobalStore`	(Optional) Boolean that indicates whether this operation occurs in a global or user-based SmartStore database. Defaults to `false`.
`storeConfig`	(Optional) `StoreConfig` object that specifies a store name and whether the store is global or user-based.
`soupName`	String. Pass in the name of the soup.
`entryIdsOrQuerySpec`	Array or QuerySpec object. Pass in the name of the soup.

Android Native Apps

Android native methods for removing entries give you the option of either handling the transaction yourself, or letting the method handle the transaction transparently. If you set the `handleTx` argument to `false`, you're responsible for starting the transaction before the call and ending it afterwards. If you use the overload that doesn't include `handleTx`, or if you set `handleTx` to false, Mobile SDK handles the transaction for you.

To remove entries by ID array in Android native apps, call either of the following methods:

```
public void delete(String soupName, Long... soupEntryIds)
public void delete(String soupName, Long[] soupEntryIds, boolean handleTx)
```

To remove entries by query in Android native apps, call either of the following methods:

```
public void deleteByQuery(String soupName, QuerySpec querySpec)
public void deleteByQuery(String soupName, QuerySpec querySpec, boolean handleTx)
```

iOS Native Apps

To remove entries by ID array in iOS native apps, call one of these methods:

Objective-C:

```
- (void)removeEntries:(NSArray*)entryIds fromSoup:(NSString*)soupName error:(NSError
**)error;
```

Swift:

```
public func remove(entryIds: [Any], forSoupName: String) -> Void
```

Example:

```
remove(entryIds: entries, forSoupNamed: soupName)
```

To remove entries by query in iOS native apps, call one of these methods:

Objective-C:

```
- (void)removeEntriesByQuery:(SFQuerySpec*)querySpec
                    fromSoup:(NSString*)soupName;
- (void)removeEntriesByQuery:(SFQuerySpec*)querySpec
                    fromSoup:(NSString*)soupName
                       error:(NSError **)error;
```

Swift:

```
func remove(usingQuerySpec: QuerySpec, entryIds: [Any], forSoupNamed: String) -> Void
```

Example:

```
var removed = removeEntries(usingQuerySpec: querySpec, forSoupNamed: soupName)
```

Managing Soups

SmartStore provides utility functionality that lets you retrieve soup metadata and perform other soup-level operations. This functionality is available for hybrid, React Native, Android native, and iOS native apps.

iOS Native Apps

To use soup management APIs in a native iOS app, import `SmartStore/SFSmartStore.h`. You call soup management methods on a shared instance of the `SmartStore` object. Obtain the shared instance by using one of the following `SFSmartStore` class methods.

Using the SmartStore instance for the current user:

Swift

```
func shared(withName: String) -> SmartStore
```

Example:

```
var store = SmartStore.shared(withName: storeName)
```

Objective-C

```
+ (id)sharedStoreWithName:(NSString*)storeName;
```

Using the SmartStore instance for a specified user:

Swift

```
func shared(withName: String, forUserAccount: SFUserAccount) -> SmartStore
```

Example:

```
var store = SmartStore.shared(withName: storeName, forUserAccount: user)
```

Objective-C

```
+ (id)sharedStoreWithName:(NSString*)storeName
                     user:(SFUserAccount *)user;
```

Example:

```
self.store = [SFSmartStore sharedStoreWithName:kDefaultSmartStoreName];
if ([self.store soupExists:@"Accounts"]) {
    [self.store removeSoup:@"Accounts"];
}
```

Android Native Apps

To use soup management APIs in a native Android app, you call methods on the shared SmartStore instance:

```
SmartStore smartStore =
    SmartStoreSDKManager.getInstance().getSmartStore();
smartStore.clearSoup("user1Soup");
```

Hybrid Apps

Each soup management function in JavaScript takes two callback functions: a success callback that returns the requested data, and an error callback. Success callbacks vary according to the soup management functions that use them. Error callbacks take a single argument, which contains an error description string. For example, you can define an error callback function as follows:

```
function(e) { alert("ERROR: " + e);}
```

To call a soup management function in JavaScript, first invoke the Cordova plug-in to initialize the SmartStore object. You then use the SmartStore object to call the soup management function. The following example defines named callback functions discretely, but you can also define them inline and anonymously.

```
var sfSmartstore = function() {
    return cordova.require("com.salesforce.plugin.smartstore");};

function onSuccessRemoveSoup(param) {
    logToConsole()("onSuccessRemoveSoup: " + param);
    $("#div_soup_status_line").html("Soup removed: "
        + SAMPLE_SOUP_NAME);
}

function onErrorRemoveSoup(param) {
    logToConsole()("onErrorRemoveSoup: " + param);
    $("#div_soup_status_line").html("removeSoup ERROR");
}
```

```
sfSmartstore().removeSoup(SAMPLE_SOUP_NAME,
     onSuccessRemoveSoup,
     onErrorRemoveSoup);
```

IN THIS SECTION:

SEE ALSO:

Adding SmartStore to Existing Android Apps

Get the Database Size

To query the amount of disk space consumed by the database, call the database size method.

Hybrid Apps

In hybrid apps, call:

```
navigator.smartstore.getDatabaseSize(successCallback, errorCallback)
```

The success callback supports a single parameter that contains the database size in bytes. For example:

```
function(dbSize) { alert("db file size is:" + dbSize + " bytes"); }
```

Android Native Apps

```
public int getDatabaseSize ()
```

iOS Native Apps

Objective-C:

```
- (long)getDatabaseSize
```

In Swift, use the Objective-C method.

Clear a Soup

To remove all entries from a soup, call the soup clearing method.

Hybrid Apps

In hybrid apps, call:

```
navigator.smartstore.clearSoup(soupName, successCallback, errorCallback)
```

The success callback supports a single parameter that contains the soup name. For example:

```
function(soupName) { alert("Soup " + soupName + " was successfully emptied."); }
```

Android Apps

In Android apps, call:

```
public void clearSoup ( String soupName )
```

iOS Apps

Objective-C:

```
- (void)clearSoup:(NSString*)soupName;
```

Swift:

```
func clearSoup(soupName:) -> Void
```

Example:

```
store.clearSoup(soupName: soupName)
```

Retrieve a Soup's Index Specs

If you want to examine or display the index specifications for a soup, call the applicable index specs retrieval method.

Hybrid Apps

In hybrid apps, call:

```
getSoupIndexSpecs()
```

In addition to the success and error callback functions, this function takes a single argument, `soupName`, which is the name of the soup. For example:

```
navigator.smartstore.getSoupIndexSpecs(soupName, successCallback,
    errorCallback)
```

The success callback supports a single parameter that contains the array of index specs. For example:

```
function(indexSpecs) { alert("Soup " + soupName +
    " has the following indexes:" + JSON.stringify(indexSpecs); }
```

Android Apps

```
public IndexSpec [] getSoupIndexSpecs ( String soupName )
```

iOS Apps

Objective-C:

```
- (NSArray*)indicesForSoup:(NSString*)soupName
```

Swift:

```
func indices(forSoupNamed: String) -> [SoupIndex]
```

Example:

```
var soupIndices = store.indices(forSoupNamed:name)
```

Change Existing Index Specs on a Soup

To change existing index specs, call the applicable soup alteration method.

Keep these important performance tips in mind when reindexing data:

- The `reIndexData` argument is optional, because reindexing can be expensive. When `reIndexData` is set to false, expect your throughput to be faster by an order of magnitude.
- Altering a soup that already contains data can degrade your app's performance. Setting `reIndexData` to true worsens the performance hit.
- As a performance guideline, expect the `alterSoup()` operation to take one second per 1000 records when `reIndexData` is set to true. Individual performance varies according to device capabilities, the size of the elements, and the number of indexes.
- `alterSoup()` and `reindexSoup()` perform better for conversion to, or creation of, JSON1 index specs than for other index spec types.
- Insert performance tends to be faster with JSON1 index specs.
- Database size is smaller with JSON1 index specs.
- Query performance is typically unaffected by JSON1 index specs.
- Other SmartStore operations must wait for the soup alteration to complete.
- If the operation is interrupted—for example, if the user exits the application—the operation automatically resumes when the application reopens the SmartStore database.

Changing Index Specs with External Storage

If you've registered a soup to use the external storage feature, use the `alterSoup` methods described in Alter a Soup with External Storage.

Hybrid Apps

In hybrid apps, call:

```
navigator.smartstore.alterSoup(soupName, indexSpecs, reIndexData,
    successCallback, errorCallback)
```

In addition to success and error callbacks, this function takes the following arguments:

Table 6: Parameters

Parameter Name	Argument Description
`soupName`	String. Pass in the name of the soup.
`indexSpecs`	Array. Pass in the set of index entries in the index specification.
`reIndexData`	Boolean. Indicate whether you want the function to re-index the soup after replacing the index specifications.

The success callback supports a single parameter that contains the soup name. For example:

```
function(soupName) { alert("Soup " + soupName +
    " was successfully altered"); }
```

The following example demonstrates a simple soup alteration. To start, the developer defines a soup that's indexed on `name` and `address` fields, and then upserts an agent record.

```
navigator.smartstore.registerSoup("myAgents",
    [{path:'name', type:'string'},
     {path:'address', type:'string'}]);
navigator.smartstore.upsertSoupEntries("myAgents",
    [{name:'James Bond',
      address:'1 market st',
      agentNumber:"007"}]);
```

When time and experience show that users really wanted to query their agents by "agentNumber" rather than `address`, the developer decides to drop the index on `address` and add an index on `agentNumber`.

```
navigator.smartstore.alterSoup("myAgents", [{path:'name',type:'string'}, {path:'agentNumber',
 type:'string'}], true);
```

Note: If the developer sets the `reIndexData` parameter to false, a query on `agentNumber` does not find the already inserted entry ("James Bond"). However, you can query that record by `name`. To support queries by `agentNumber`, you'd first have to call `navigator.smartstore.reIndexSoup("myAgents", ["agentNumber"])`

Android Native Apps

In an Android native app, call:

```
public void alterSoup(String soupName, IndexSpec [] indexSpecs, boolean reIndexData) throws
 JSONException
```

iOS Native Apps

Objective-C:

```
- (BOOL) alterSoup:(NSString*)soupName withIndexSpecs:(NSArray*)indexSpecs
reIndexData:(BOOL)reIndexData;
```

In Swift, use the Objective-C method.

Reindex a Soup

Use reindexing if you previously altered a soup without reindexing the data, but later you want to make sure all elements in the soup are properly indexed. Both `alterSoup()` and `reindexSoup()` perform better for conversion to, or creation of, JSON1 index specs than for other index spec types.

Hybrid Apps

In hybrid apps, call:

```
navigator.smartstore.reIndexSoup(soupName, listOfPaths, successCallback, errorCallback)
```

In addition to the success and error callback functions, this function takes a single argument, `soupName`, which is the name of the soup. For example:this function takes additional arguments:

Parameter Name	Argument Description
`soupName`	String. Pass in the name of the soup.
`listOfPaths`	Array. List of index paths on which you want to re-index.

The success callback supports a single parameter that contains the soup name. For example:

```
function(soupName) { alert("Soup " + soupName +
    " was successfully re-indexed."); }
```

Android Apps

In Android apps, call:

```
public void reIndexSoup(String soupName, String[] indexPaths, boolean handleTx)
```

iOS Apps

Objective-C:

```
- (BOOL) reIndexSoup:(NSString*)soupName
      withIndexPaths:(NSArray*)indexPaths
```

In Swift, use the Objective-C method.

Remove a Soup

Removing a soup deletes it. When a user signs out, all soups are deleted automatically. For other occasions in which you want to delete a soup, call the applicable soup removal method.

Hybrid Apps

In hybrid apps, call:

```
navigator.smartstore.removeSoup(soupName,successCallback,errorCallback);
```

Android Apps

In Android apps, call:

```
public void dropSoup ( String soupName )
```

iOS Apps

Objective-C:

```
- (void)removeSoup:(NSString*)soupName
```

Swift:

```
func removeSoup(soupName: String) -> Void
```

Example:

```
store.removeSoup(soupName: soupName)
```

Managing Stores

If you create global stores, you're required to perform cleanup when the app exits. Also, if you create multiple user stores, you can perform cleanup if you're no longer using particular stores. SmartStore provides methods deleting named and global stores. For hybrid apps, SmartStore also provides functions for getting a list of named stores.

iOS Native Apps

Mobile SDK for iOS defines the following `SFSmartStore` methods for removing stores.

Swift

```
func removeShared(withName: String) -> Void
func removeShared(withName: String, forUserAccount: UserAccount) -> Void
```

```
func removeSharedGlobal(withName: String) -> Void
func removeAllForCurrentUser() -> Void
func removeAll(forUserAccount: UserAccount) -> Void
func removeAllGlobal() -> Void
```

Objective-C

```
+ (void)removeSharedStoreWithName:(NSString *)storeName;
+ (void)removeSharedStoreWithName:(NSString *)storeName forUser:(SFUserAccount *)user;
+ (void)removeSharedGlobalStoreWithName:(NSString *)storeName;
+ (void)removeAllStores;
+ (void)removeAllStoresForUser:(SFUserAccount *)user;
+ (void)removeAllGlobalStores;
```

In addition, SmartStore provides the following methods for retrieving store names. Use this method for both Swift and Objective-C apps.

```
+ (NSArray *)allStoreNames;
+ (NSArray *)allGlobalStoreNames;
```

Android Native Apps

Mobile SDK for Android defines the following `SmartStoreSDKManager` methods for removing stores.

```
public void removeGlobalSmartStore(String dbName)
public void removeSmartStore()
public void removeSmartStore(UserAccount account)
public void removeSmartStore(UserAccount account, String communityId)
public void removeSmartStore(String dbNamePrefix, UserAccount account, String communityId)
```

In addition, SmartStore provides the following methods for retrieving store names.

```
public List<String> getGlobalStoresPrefixList()
public List<String> getUserStoresPrefixList()
```

Hybrid Apps

SmartStore defines the following functions for removing stores. Each function takes success and error callbacks. The `removeStore()` function also requires either a `StoreConfig` object that specifies the store name, or just the store name as a string.

```
removeStore(storeConfig,successCB, errorCB)
removeAllGlobalStores(successCB, errorCB)
removeAllStores(successCB, errorCB)
```

In addition, the hybrid version of SmartStore provides the following functions for retrieving the `StoreConfig` objects for defined stores.

```
getAllStores(successCB, errorCB)
getAllGlobalStores(successCB, errorCB)
getAllStores(successCB, errorCB)
getAllGlobalStores(successCB, errorCB)
```

Example: **Removing All SmartStore Data at Runtime**

Sometimes an app must remove all data in a store without logging out the current user. In this case, keep in mind that the sync manager object sets up a table to track syncs. If you delete this table, the manager can't continue. Therefore, the recommended way to reset SmartStore to a zero-data state is as follows:

1. Drop the sync managers associated with the current user.

   iOS

   Swift

   Call the following `SyncManager` method:

   ```
   func removeSharedInstance(user: UserAccount) -> Void
   ```

   Objective-C

   Call the following `SFMobileSyncSyncManager` method:

   ```
   + (void)removeSharedInstance:(SFUserAccount*)user;
   ```

   Android

   Call the following `MobileSyncSyncManager` method:

   ```
   public static synchronized void reset(UserAccount account)
   ```

2. Drop the stores associated with the current user.

   iOS

   Swift

   Call the following `SmartStore` method:

   ```
   func removeAll(forUserAccount: UserAccount) -> Void
   ```

   Objective-C

   Call the following `SFSmartStore` method:

   ```
   + (void)removeAllStoresForUser:(SFUserAccount *)user;
   ```

   Android

   Call the following `SmartStoreSDKManager` method:

   ```
   public void removeAllUserStores()
   ```

Testing with the SmartStore Inspector

Verifying SmartStore operations during testing can become a tedious and time-consuming effort. SmartStore Inspector comes to the rescue.

During testing, it's helpful to see if your code is handling SmartStore data as you intended. The SmartStore Inspector provides a mobile UI class for that purpose. With the SmartStore Inspector you can:

- Examine soup metadata, such as soup names and index specs for any soup
- Clear a soup's contents
- Perform Smart SQL queries

Note: SmartStore Inspector is for testing and debugging only. If you add code references to SmartStore Inspector, be sure to remove them before you build the final version of your app.

As of Mobile SDK 6.0, you can access SmartStore Inspector in debug builds from the Dev Tools menu. This feature no longer requires you to add code to your app. See Mobile SDK Tools for Developers.

Using the Mock SmartStore

To facilitate developing and testing code that makes use of the SmartStore while running outside the container, you can use an emulated SmartStore.

MockSmartStore is a JavaScript implementation of SmartStore that stores data in local storage (or optionally just in memory).

In the `external/shared/test` directory, you'll find the following files:

- `MockCordova.js`—A minimal implementation of Cordova functions intended only for testing plug-ins outside the container. Intercepts Cordova plug-in calls.
- `MockSmartStore.js`—A JavaScript implementation of SmartStore intended only for development and testing outside the container. Also intercepts SmartStore Cordova plug-in calls and handles them using a MockSmartStore.

When you're developing an application using SmartStore, make the following changes to test your app outside the container:

- Include `MockCordova.js` instead of `cordova.js`.
- Include `MockSmartStore.js`.

To see a MockSmartStore example, check out `test/test.html` in the github.com/forcedotcom/SalesforceMobileSDK-Shared repo.

Same-Origin Policies

Same-origin policy permits scripts running on pages originating from the same site to access each other's methods and properties with no specific restrictions; it also blocks access to most methods and properties across pages on different sites. Same-origin policy restrictions are not an issue when your code runs inside the container, because the container disables same-origin policy in the webview. However, if you call a remote API, you need to worry about same-origin policy restrictions.

Fortunately, browsers offer ways to turn off same-origin policy, and you can research how to do that with your particular browser. If you want to make XHR calls against Salesforce Platform from JavaScript files loaded from the local file system, you should start your browser with same-origin policy disabled. The following article describes how to disable same-origin policy on several popular browsers: Getting Around Same-Origin Policy in Web Browsers.

Authentication

For authentication with MockSmartStore, you will need to capture access tokens and refresh tokens from a real session and hand code them in your JavaScript app. You'll also need these tokens to initialize the `force.js` JavaScript toolkit.

Note:

- MockSmartStore doesn't encrypt data and is not meant to be used in production applications.
- MockSmartStore currently supports the following forms of Smart SQL queries:
  - `SELECT...WHERE...`. For example:

```
SELECT {soupName:selectField} FROM {soupName} WHERE {soupName:whereField} IN
(values)
```

- `SELECT...WHERE...ORDER BY...`. For example:

```
SELECT {soupName:_soup} FROM {soupName} WHERE {soupName:whereField} LIKE 'value'
 ORDER BY LOWER({soupName:orderByField})
```

- `SELECT count(*) FROM {soupName}`

MockSmartStore doesn't directly support the simpler types of Smart SQL statements that are handled by the `build*QuerySpec()` functions. Instead, use the query spec function that suits your purpose.

SEE ALSO:

Retrieving Data from a Soup

Preparing Soups for Mobile Sync

Soups that exchange information with the Salesforce cloud typically use Mobile Sync for synchronization. To support Mobile Sync, most app types require you to create and manage special soup fields for "sync up" operations.

Types of apps that require you to code these special fields include:

- Hybrid apps that do not use `Force.SObject` (from `mobilesync.js`) to create and manage local records
- Native apps
- React Native apps

If your hybrid app uses `Force.SObject` for local records, Mobile Sync automatically creates and manages these fields for you. You can ignore the rest of this discussion.

Add Required Fields

1. Add the following fields to your soup elements. The first three are operation type fields:

Operation Type Fields

Be sure to set the appropriate field to true for every create, update, or delete operation.

`__locally_created__`

- Type: `string`
- Set this field to true on elements that your app *creates* locally.

`__locally_updated__`

- Type: `string`
- Set this field to true after your app *updates* an element locally.

`__locally_deleted__`

- Type: `string`
- Set this field to true when your app *is deleting* an element locally.

Control Fields

`__local__`

- Type: `string`
- This field indicates that some local change has occurred. You're required to:

- Set this field to true when any of the operation type fields is true.
- Add a string index spec on this field.

`__sync_id__`

- Type: `integer`
- This field ensures that the `cleanResyncGhosts()` method removes only the desired soup elements. Mobile Sync manages the content of this field for you.

2. Add a soup index for each of the operation and control fields. See Registering Soups with Configuration Files.

Mobile Sync Behavior

During sync up operations, Mobile Sync looks for soup elements with `__local__` set to true. For each match, it evaluates the operation type fields and then performs the operation indicated by the following precedence hierarchy.

Precedence	Field	If set to true...
1 (highest)	`__locally_deleted__`	• `__locally_created__` and `__locally_updated__` flags are ignored. • Mobile Sync deletes the local record and, if it exists, the server record. If the server record does not exist, no remote action occurs.
2	`__locally_created__`	• `__locally_updated__` flag is ignored. • If `__locally_deleted__` is not true, Mobile Sync creates the record on the server.
3	`__locally_updated__`	• Ignored if either `__locally_deleted__` or `__locally_created__` is true. • Otherwise, Mobile Sync writes the updated record to the server.

Finally, Mobile Sync resets all four fields to false.

Example: The following examples are taken from the various language versions of the MobileSyncExplorer sample app.

iOS Native

This Objective-C example sets system fields by sending `updateSoupForFieldName:fieldValue:` messages to an `SObjectData` object. Using `SFMobileSyncSyncManager` constants for the field names, it sets the `__local__` and `__locally_created__` fields before upserting the new element. You can find the `SObjectData` definition in the iOS sample app.

```
- (void)createLocalData:(SObjectData *)newData {
    [newData updateSoupForFieldName:kSyncManagerLocal fieldValue:@YES];
    [newData updateSoupForFieldName:kSyncManagerLocallyCreated fieldValue:@YES];
    [self.store upsertEntries:@[ newData.soupDict ] toSoup:[[newData class]
dataSpec].soupName];
}
```

Android Native

The following Java example handles created and updated elements, but not deletions. It calls the `JSONObject put()` method to create and initialize the system fields, using `SyncManager` constants for the field names. After the fields are properly assigned, it either creates or upserts the element based on the `isCreate` control flag.

```
contact.put(SyncTarget.LOCAL, true);
contact.put(SyncTarget.LOCALLY_UPDATED, !isCreate);
contact.put(SyncTarget.LOCALLY_CREATED, isCreate);
contact.put(SyncTarget.LOCALLY_DELETED, false);
if (isCreate) {
    smartStore.create(ContactListLoader.CONTACT_SOUP, contact);
} else {
    smartStore.upsert(ContactListLoader.CONTACT_SOUP, contact);
}
```

Hybrid with the Mobile Sync Plug-in and React Native

The following React Native code can easily be adapted for hybrid apps that use the Mobile Sync plug-in. This example shows how to update and delete—or undelete—a contact. The `onSaveContact()` function marks the record as updated, sets `__local__` to true, and then saves the changes. The `onDeleteUndeleteContact()` function flips the `__locally_deleted__` field. It then sets the `__local__` field to match the operation type value and saves the changes.

The `storeMgr` object is defined in the sample project as a wrapper around SmartStore and the Mobile Sync plug-in. Its `saveContact()` function accepts a contact object and a callback, and upserts the contact into the soup. The callback shown here calls `navigator.pop()`, which is specific to React Native. Hybrid apps can replace the `saveContact()` function with any code that calls the SmartStore `upsert()` function.

```
onSaveContact: {
    const contact = this.state.contact;
    contact.__locally_updated__ = contact.__local__ = true;
    storeMgr.saveContact(contact, () => {navigator.pop();});
},

onDeleteUndeleteContact: {
    const contact = this.state.contact;
    contact.__locally_deleted__ = !contact.__locally_deleted__;
    contact.__local__ = contact.__locally_deleted__ || contact.__locally_updated__ ||
 contact.__locally_created__;
    storeMgr.saveContact(contact, () => {navigator.pop();});
},
```

Using SmartStore in Swift Apps

You can easily install the basic plumbing for SmartStore in a forceios native Swift project.

In this example, you create a SmartStore soup and upsert the queried list of contact names into that soup. You then change the Swift template app flow to populate the table view from the soup instead of directly from the REST response. If you're not familiar with Xcode project structure, consult the *Xcode Help*.

1. Using forceios, create a native Swift project similar to the following example:

```
$ forceios create
Enter your application type (native_swift or native, leave empty for native_swift):
<Press RETURN>
Enter your application name: <Enter any name you like>
Enter your package name: com.myapps.ios
```

```
Enter your organization name (Acme, Inc.): MyApps.com
Enter output directory for your app (leave empty for the current directory): <Press
RETURN or enter a directory name>
```

2. In your project's root directory, create a `userstore.json` file with the following content.

```
{ "soups": [
    {
    "soupName": "Contact",
    "indexes": [
        { "path": "Name", "type": "string"},
        { "path": "Id", "type": "string"}
        ]
    }
]}
```

3. Open your app's `.xcworkspace` file in Xcode.
4. Add your configuration file to your project.
   a. In the Xcode Project navigator, select the project node.
   b. In the Editor window, select **Build Phases**.
   c. Expand **Copy Bundle Resources**.
   d. Click **+** ("Add items").
   e. Select your soup configuration file. If your file is not already in an Xcode project folder:
      i. To select your file in Finder, click **Add Other...**.
      ii. Click **Open**, then click **Finish**.
5. In your project's source code folder, select `Classes/AppDelegate.swift`.
6. In the `application(_:didFinishLaunchingWithOptions:)` callback method, load `userstore.json` definitions in the call to `AuthHelper.loginIfRequired`.

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool
{
    self.window = UIWindow(frame: UIScreen.main.bounds)
    self.initializeAppViewState();
    ...

    AuthHelper.loginIfRequired { [weak self] in
        MobileSyncSDKManager.shared.setupUserStoreFromDefaultConfig()
        self?.setupRootViewController()
    }
    return true
}
```

Your app is now set up to load your SmartStore configuration file at startup. This action creates the soups you specified as empty tables. Let's configure the `RootViewController` class to use SmartStore.

7. In `RootViewController.swift`, import `SmartStore`:

```
import SmartStore
```

8. At the top of the `RootViewController` class, declare a variable for a `SmartStore` instance.

```
class RootViewController : UITableViewController
{
    var dataRows = [NSDictionary]()
    var store = SmartStore.shared(withName: SmartStore.defaultStoreName)
```

9. On the next line, declare a constant that defines an `OSLog` component.

```
class RootViewController : UITableViewController
{
    var dataRows = [NSDictionary]()
    var store = SmartStore.shared(withName: SmartStore.defaultStoreName)
    let mylog = OSLog(subsystem: "com.testapp.swift", category: "tutorial")
```

10. In the `loadView()` method, find the call to `.query` and add the `Id` field to the SOQL statement.

```
let request = RestClient.shared.request(forQuery: "SELECT Name, Id FROM Contact LIMIT
10")
```

11. In the `handleSuccess(_:_:)` method, immediately after the `guard` block, add the following code.

```
func handleSuccess(response: RestResponse, request: RestRequest) {
    guard let jsonResponse  = try? response.asJson() as? [String:Any],
        let records = jsonResponse["records"] as? [[String:Any]]  else {
        SalesforceLogger.d(RootViewController.self, message:"Empty Response for :
\(request)")
        return
    }

    if let smartstore = self.store,
       smartstore.soupExists(forName: "Contact") {
            smartstore.clearSoup("Contact")
            smartstore.upsert(entries: records, forSoupNamed: "Contact")
            os_log("\nSmartStore loaded records.", log: self.mylog, type: .debug)
       }


    SalesforceLogger.d(type(of:self), message:"Invoked: \(request)")
    DispatchQueue.main.async {
        self.dataRows = records
        self.tableView.reloadData()
    }
}// end of handleSuccess method
```

This code checks whether the Contact soup exists. If the soup exists, the code clears all data from the soup, and then upserts the retrieved records.

12. Launch the app, then check your work using the Dev Tools menu.
    a. To bring up the menu, type *control + command + z* if you're using the iOS emulator, or shake your iOS device.
    b. Click **Inspect SmartStore**.
    c. To list your Contact soup and number of records, click **Soups**.

    Note: If you get a "Query: No soups found" message, chances are you have an error in your `userstore.json` file.

You've now created and populated a SmartStore soup. However, at this point your soup doesn't actually serve a purpose. Let's make it more useful by populating the list view from SmartStore records rather than directly from the REST response.

1. After the `handleSuccess(_:_:)` method, add a method named `loadFromStore()`.

```
func loadFromStore() {

}
```

2. In `loadFromStore()`, define an `if` block that builds a Smart SQLquery specification as its first condition. Configure the query to extract the first 10 Name values from the Contact soup.

```
func loadFromStore() {
    if let querySpec = QuerySpec.buildSmartQuerySpec(
        smartSql: "select {Contact:Name} from {Contact}", pageSize: 10),

}
```

3. Add a second condition that verifies the SmartStore handle and a third condition that runs the SmartStore query. Since the `query` method throws an exception, call it from a `do...try...catch` block.

```
func loadFromStore() {
    if let querySpec = QuerySpec.buildSmartQuerySpec(
        smartSql: "select {Contact:Name} from {Contact}", pageSize: 10),
        let smartStore = self.store,
        let records = try? smartStore.query(using: querySpec,
            startingFromPageIndex: 0) as? [[String]] {

    }

}
```

4. Transfer the names returned by the SmartStore query to the view's `dataRows` member .

```
func loadFromStore() {
    if let querySpec = QuerySpec.buildSmartQuerySpec(
        smartSql: "select {Contact:Name} from {Contact}", pageSize: 10),
        let smartStore = self.store,
        let records = try? smartStore.query(using: querySpec,
            startingFromPageIndex: 0) as? [[String]] {
        self.dataRows = records.map({ row in
            return ["Name": row[0]]
        })
    }
}
```

5. Using the `DispatchQueue` system object, switch to the main thread and refresh the view's displayed data.

```
func loadFromStore() {
    if let querySpec = QuerySpec.buildSmartQuerySpec(
        smartSql: "select {Contact:Name} from {Contact}", pageSize: 10),
        let smartStore = self.store,
        let records = try? smartStore.query(using: querySpec,
            startingFromPageIndex: 0) as? [[String]] {
        self.dataRows = records.map({ row in
```

```
            return ["Name": row[0]]
        })
        DispatchQueue.main.async {
            self.tableView.reloadData()
        }
    }
}
```

6. Scroll back to the `handleSuccess(_:_:)` method and remove the existing code that reloads the view's data.

```
func handleSuccess(response: RestResponse, request: RestRequest) {
    guard let jsonResponse  = try? response.asJson() as? [String:Any],
        let records = jsonResponse["records"] as? [[String:Any]]  else {
        SalesforceLogger.d(RootViewController.self, message:"Empty Response for :
\(request)")
        return
    }

    if ((self.store.soupExists(forName: "Contact"))) {
        self.store.clearSoup("Contact")
        self.store.upsert(entries: records, forSoupNamed: "Contact")
        os_log("\nSmartStore loaded records.", log: self.mylog, type: .debug)
    }

    // Remove the following lines
    SalesforceLogger.d(type(of:self), message:"Invoked: \(request)")
    DispatchQueue.main.async {
        self.dataRows = records
        self.tableView.reloadData()
    }
}// end of handleSuccess method
```

7. Using `self`, call your new `loadFromStore()` method immediately after the `upsert(entries:forSoupNamed:)` call.

```
func handleSuccess(response: RestResponse, request: RestRequest) {
    guard let jsonResponse  = try? response.asJson() as? [String:Any],
        let records = jsonResponse["records"] as? [[String:Any]]  else {
        SalesforceLogger.d(RootViewController.self, message:"Empty Response for :
\(request)")
        return
    }

    if ((self.store.soupExists(forName: "Contact"))) {
        self.store.clearSoup("Contact")
        self.store.upsert(entries: records, forSoupNamed: "Contact")
        self.loadFromStore()
        os_log("\nSmartStore loaded records.", log: self.mylog, type: .debug)
    }
} // end of loadView
```

When you retest your app, you see that the table view is populated as before, but from SmartStore rather than a live REST response. In the real world, you'd create an editing interface for the Contact list, and then upsert your customers' edits to SmartStore. The customer

could then continue working on the Contact list even if the mobile device lost connectivity. When connectivity is restored, you could then merge the customer's work to the server—and also resync SmartStore—using Mobile Sync.

Using Mobile Sync to Access Salesforce Objects

Mobile Sync enables mobile customers whose devices have lost connectivity to continue working on Salesforce data. When connectivity is restored, Mobile Sync synchronizes any changes made to local SmartStore records with the corresponding records on the Salesforce server.

Mobile Sync predefines cache policies for fine-tuning interactions between cached data and server data in offline and online scenarios. Mobile SDK convenience methods automate common network activities—fetching sObject metadata, fetching a list of most recently used objects, and building SOQL and SOSL queries.

Using Mobile Sync in Native Apps

The native Mobile Sync library provides native iOS and Android APIs that simplify the development of offline-ready apps. A subset of this native functionality is also available to hybrid apps through a Cordova plug-in.

Mobile Sync libraries offer parallel architecture and functionality for iOS and Android, expressed in each platform's native language. The shared functional concepts are straightforward:

- Query Salesforce object metadata by calling Salesforce REST APIs.
- Store the retrieved object data locally and securely for offline use.
- Sync data changes when the device goes from an offline to an online state.

With Mobile Sync native libraries, you can:

- Get and post data by interacting with a server endpoint. Mobile Sync helper APIs encode the most commonly used endpoints. These APIs help you fetch sObject metadata, retrieve the list of most recently used (MRU) objects, and build SOQL and SOSL queries. You can also use arbitrary endpoints that you specify in a custom class.
- Fetch Salesforce records and metadata and cache them on the device, using one of the pre-defined cache policies.
- Edit records offline and save them offline in SmartStore.
- Synchronize batches of records by pushing locally modified data to the Salesforce cloud.

Mobile Sync Components

The following components form the basis of Mobile Sync architecture.

Sync Manager Class

- **iOS class:**

Swift	Objective-C
`SyncManager`	`SFMobileSyncSyncManager`

- **Android class:** `com.salesforce.androidsdk.mobilesync.manager.SyncManager`

  Provides APIs for synchronizing large batches of sObjects between the server and SmartStore. This class works independently of the metadata manager and is intended for the simplest and most common sync operations. Sync managers can "sync down"—download sets of sObjects from the server to SmartStore—and "sync up"—upload local sObjects to the server.

The sync manager works in tandem with the following utility classes:

Sync State Class
Tracks the state of a sync operation. States include:

- New—The sync operation has been initiated but has not yet entered a transaction with the server.
- Running—The sync operation is negotiating a sync transaction with the server.
- Done—The sync operation finished successfully.
- Failed—The sync operation finished unsuccessfully.
- **iOS:**

Swift	Objective-C
`SyncState`	`SFSyncState`

- **Android:** `com.salesforce.androidsdk.mobilesync.util.SyncState`

Sync Target Class
Parent class for specifying the sObjects to be downloaded during a "sync down" operation.

- **iOS:**

Swift	Objective-C
`SyncTarget`	`SFSyncTarget`

- **Android:** `com.salesforce.androidsdk.mobilesync.util.SyncTarget`

Sync Options Class
Specifies configuration options for a "sync up" operation. Options include the list of field names to be synced.

- **iOS:**

Swift	Objective-C
`SyncOptions`	`SFSyncOptions`

- **Android:** `com.salesforce.androidsdk.mobilesync.util.SyncOptions`

SOQL Builder
Utility class that makes it easy to build a SOQL query statement, by specifying the individual query clauses.

- **iOS class:**

Swift	Objective-C
`SFSDKSoqlBuilder`	`SFSDKSoqlBuilder`

- **Android class:** `com.salesforce.androidsdk.mobilesync.util.SOQLBuilder`

SOSL Builder
Utility class that makes it easy to build a SOSL query statement, by specifying the individual query clauses.

- **iOS class:**

Swift	Objective-C
`SFSDKSoslBuilder`	`SFSDKSoslBuilder`

- **Android class:** `com.salesforce.androidsdk.mobilesync.util.SOSLBuilder`

MobileSyncSDKManager
Beginning in Mobile SDK 6.0, all forcedroid and forceios template apps use `MobileSyncSDKManager` as the base SDK entry point. The class name, `MobileSyncSDKManager`, is the same for iOS (Objective-C and Swift) and Android. In Android, your `App` class extends `MobileSyncSDKManager` instead of `SalesforceSDKManager`. In iOS, the `init` method of your `AppDelegate` class uses a shared instance of `MobileSyncSDKManager` instead of `SalesforceSDKManager`. This change applies to both native and hybrid apps.

Note: To support multi-user switching, Mobile Sync creates unique instances of its components for each user account.

SEE ALSO:
SDK Manager Classes
SalesforceSDKManager Class

Creating Native Apps with Mobile Sync

In forceios and forcedroid version 5.0 and later, generating native Mobile Sync apps literally requires no extra effort. Any native app you create automatically includes the SmartStore and Mobile Sync libraries.

Adding Mobile Sync to Existing Android Apps

The following steps show you how to add Mobile Sync to an existing Android project (hybrid or native) created with Mobile SDK 4.0 or later.

1. If your app is currently built on Mobile SDK 3.3 or earlier, upgrade your project to the latest SDK version as described in Migrating from the Previous Release.
2. Add the Mobile Sync library project to your project. Mobile Sync uses SmartStore, so you also need to add that library if your project wasn't originally built with SmartStore.
   a. In Android Studio, add the `libs/MobileSync` project to your module dependencies.
3. Throughout your project, change all code that uses the `SalesforceSDKManager` object to use `MobileSyncSDKManager` instead.

   Note: If you do a project-wide search and replace, be sure *not* to change the `KeyInterface` import, which should remain

   ```
   import com.salesforce.androidsdk.app.SalesforceSDKManager.KeyInterface;
   ```

Adding Mobile Sync to Existing iOS Apps

You can easily upgrade existing iOS projects to support Mobile Sync: Just use forceios to create a new project, then add in your assets. However, if you'd like to know the steps for upgrading older Mobile Sync apps to Mobile SDK 6.0 or later, you're in the right place.

In Mobile SDK 4.0, Mobile Sync moved out of Mobile SDK core into its own library. Mobile Sync relies on SmartStore, so forceios automatically adds SmartStore to your project. In addition, native iOS projects in Mobile SDK 6.0 and later require a `MobileSyncSDKManager` object to initialize the app.

Instead of making the updates piece by piece, we recommend that you create a new native project, then copy your assets into that project. You can create the new shell project either manually using template source files, or with forceios. To create the project manually, see Creating an iOS Swift Project Manually.

The native template app uses the `MobileSyncSDKManager` class by default and imports the correct libraries for you, plus many other updates. See https://github.com/forcedotcom/SalesforceMobileSDK-Templates/blob/master/iOSNativeTemplate/iOSNativeTemplate/AppDelegate.m

About Sync Targets

Sync targets configure data transfers between the Salesforce cloud and a local database on a mobile device. Mobile SDK 5.1 enhances the capabilities of targets to give developers more control over two-way data synchronization.

Mobile Sync is all about syncing data. In essence, it

- Syncs data *down* from the server to a local database, and
- Syncs data *up* from the local database to the server.

Often, the data you're transferring doesn't cross break any rules, and the default sync targets work fine. For special cases, though, you can provide your own sync target to make sure that data transfers occur as expected. An example is when an object contains fields that are required but that apps can't update. If a sync up operation tries to upload both new and updated records with a single field list, the operation fails if it tries to update locked fields. Beginning in Mobile SDK 5.1, you have other options that can often spare you from implementing a custom target.

Decentralizing Sync Manager Tasks (Or, Power to the Custom Targets!)

In the first Mobile Sync release, the sync manager class internally handled all server and local database interactions. In addition, the sync manager was a "final" class that was off-limits for developer customization. Developers were unable to add their own nuances or extended functionality.

Later, an architectural refactoring delegated all server interactions from the sync manager class to sync down and sync up target classes. Thus began a transfer of power from the monolithic sync manager to the flexible sync targets. Unlike sync manager class, the second-generation target classes let developers subclass sync targets for their own purposes. By controlling interactions with servers, custom sync targets can talk to arbitrary server endpoints, or transform data before storing it.

Mobile SDK 5.1 enhances Mobile Sync still further by moving local database interactions into targets. This enhancement offers several benefits.

- It decouples Mobile Sync from SmartStore, giving developers the freedom to use other stores.
- It allows developers to use their own data layouts and capture local data changes however they like.
- It enables more complex objects, such as targets that can simultaneously handle multiple record types.

In short, Mobile Sync now offers developers significant control over the entire round trip of data synchronization.

Note: Beginning in Mobile SDK 5.1, additional "sync up" options can sometimes obviate the need for a custom target. See Defining a Custom Sync Up Target.

Defining Sync Names and Sync Configuration Files

Beginning in Mobile SDK 6.0, you can define sync configuration files and assign names to sync configurations. You can use sync names to run, edit, or delete a saved sync operation. Since all platforms and app types use the same configuration files, you can describe all your syncs in a single file. You can then compile that file into any Mobile SDK project.

Mobile Sync configuration files use JSON objects to express sync definitions. You can provide these files to avoid coding sync down and sync up configurations. The JSON schema for configuration files is the same for all app types and platforms. Hybrid apps load the configuration files automatically, while other apps load them with a single line of code. To keep the mechanism simple, Mobile SDK enforces the following file naming conventions:

- To define sync operations for the default global store, provide a file named `globalsyncs.json`.
- To define sync operations for the default user store, provide a file named `usersyncs.json`.

Configuration files can define syncs only in the default global store and default user store. For named stores, you define syncs through code.

In native and React Native apps, you load your JSON configuration file by calling a sync loading method. Call this method in your app initialization code after the customer successfully logs in. For example, in iOS, call this method in the block you pass to `loginIfRequired`. Call these methods only if you're using a `globalsyncs.json` or `usersyncs.json` file instead of code to configure your syncs. Don't call sync loading methods more than one time.

In hybrid apps that include them, sync configuration files are loaded automatically. To see loader examples, study the MobileSyncExplorer and MobileSyncExplorerHybrid sample apps. These apps use configuration files to set up their sync operations.

Note:

- Configuration files are intended for initial setup only. You can't change existing syncs by revising the JSON file and reloading it at runtime. Instead, you can use a `syncUp` or `syncDown` method to define the sync inline.
- If the name that a configuration file assigns to a sync operation exists, Mobile SDK ignores the configuration file. In this case, you can set up and manage your sync only through code.

Configuration File Format

The following example demonstrates the configuration file format.

```
{
    "syncs": [
        {
            "syncName": "sync1",
            "syncType": "syncDown",
            "soupName": "accounts",
            "target": {"type":"soql",
                       "query":"SELECT Id, Name, LastModifiedDate
                                FROM Account",
                       "maxBatchSize":200},
            "options": {"mergeMode":"OVERWRITE"}
        },
        {
            "syncName": "sync2",
            "syncType": "syncUp",
            "soupName": "accounts",
            "target": {"createFieldlist":["Name"]},
            "options": {"fieldlist":["Id", "Name", "LastModifiedDate"],
                        "mergeMode":"LEAVE_IF_CHANGED"}
```

```
            }
        ]
}
```

For sync down, the `"target"` property's `"type"` property accepts any one of the following values:

- **`"soql"`**
  Uses a SOQL query for sync down.

  Properties:

  - "type":"soql"
  - "query": <string>
  - "idFieldName": <string>
  - "modificationDateFieldName": <string>
  - "maxBatchSize": <integer>, any value from 200 and 2,000 (default value is 2,000)

  Required: "type", "query"

- **`"sosl"`**
  Uses a SOSL query for sync down.

  Properties:

  - "type":"sosl"
  - "query": <string>
  - "idFieldName": <string>
  - "modificationDateFieldName": <string>
  - "maxBatchSize": <integer>, any value from 200 and 2,000 (default value is 2,000)

  Required: "type", "query"

- **`"briefcase"`**
  Uses a briefcase for sync down.

  Properties:

  - "type":"briefcase"
  - "infos": array of <BriefcaseObjectInfo> items

  Required: "type", "infos"

  See Using the Briefcase Sync Down Target

- **`"mru"`**
  Syncs most recently used records for the given object.

  Properties:

  - "type":"mru"
  - "sobjectType": <string>
  - "fieldlist": array of <string> items
  - "idFieldName": <string>
  - "modificationDateFieldName": <string>

  Required: "type", "sobjectType", "fieldlist"

- **`"refresh"`**
  Refreshes a sync of the given object and fields in the given soup.

  Properties:

  - "type":"refresh"
  - "sobjectType": <string>
  - "fieldlist": array of <string> items
  - "soupName": <string>
  - "idFieldName": <string>
  - "modificationDateFieldName": <string>

  Required: "type", "sobjectType", "fieldlist", "soupName"

- **`"layout"`**
  Syncs layouts for the given object.

  Properties:

  - "type":"layout"
  - "sobjectType": <string>
  - "formFactor": *Choice*: <"Large" | "Medium" | "Small">
  - "layoutType": *Choice*: <"Compact" | "Full">
  - "mode": *Choice*: <"Create" | "Edit" | "View">
  - "recordTypeId": <string>
  - "idFieldName": <string>
  - "modificationDateFieldName": <string>

  Required: "type", "sobjectType", "layoutType"

- **`"metadata"`**
  Syncs metadata for the given object.

  Properties:

  - "type":"metadata"
  - "sobjectType": <string>
  - "idFieldName": <string>
  - "modificationDateFieldName": <string>

  Required: "type", "sobjectType"

- **`"parent_children"`**
  Syncs related records for the given parent object.

  Properties: See Syncing Related Records on page 330.

- **`"custom"`**
  Syncs using your custom sync down target. Assign the names of your native target sync down classes for iOS and Android the "iOSImpl" and "androidImpl" properties.

  Properties:

  - "type":"custom"

- "iOSImpl": <string>
- "androidImpl": <string>
- "idFieldName": <string>
- "modificationDateFieldName": <string>

Required: "type", "iOSImpl", "androidImpl"

By default, sync up targets defined in sync config files use batch APIs. In addition, all sync up targets:

- Can define a `"createFieldList"` property.
- Can define an `"updateFieldList"` property.
- Don't define a `"type"` property.

Here are the specific settings for the various types of sync up targets.

- **sObject Collection**
  Standard sync up target type for Mobile SDK 10.1 and later.

  Properties:

  - "createFieldlist": array of <string> items
  - "updateFieldlist": array of <string> items
  - "externalIdFieldName": <string>

  Required: none

- **Batch**
  Similar to the standard target, but uses smaller batch operations. Standard target for Mobile SDK 7.1 to 10.0.

  Properties:

  - "createFieldlist": array of <string> items
  - "updateFieldlist": array of <string> items
  - "externalIdFieldName": <string>

  Required: none

- **Non-batch**
  Similar to the standard target, but doesn't use batch or collection operations.

  Properties:

  - "iOSImpl":"SFSyncUpTarget"
  - "androidImpl":"com.salesforce.androidsdk.mobilesync.target.SyncUpTarget"
  - "createFieldlist": array of <string> items
  - "updateFieldlist": array of <string> items
  - "externalIdFieldName": <string>

  Required: "iOSImpl", "androidImpl"

- **Parent-child**
  Syncs related records for the given parent object.

  Properties: See Syncing Related Records on page 330.

- **Custom**
  Syncs using your custom sync up target. Assign the names of your native sync up target classes for iOS and Android to the "iOSImpl" and "androidImpl" properties.

  Properties:

  - "iOSImpl": <string>
  - "androidImpl": <string>
  - "createFieldlist": array of <string> items
  - "updateFieldlist": array of <string> items

  Required: "iOSImpl", "androidImpl"

Target JSON definitions are specified in the Mobile Sync JSON schema.

Configuration File Locations

Configuration file placement varies according to app type and platform. Mobile SDK looks for configuration files in the following locations:

iOS (Native and React Native)
Under `/` in the Resources bundle

Android (Native and React Native)
In the `/res/raw` project folder

Hybrid
In your Cordova project, do the following:

1. Place the configuration file in the top-level `www/` folder.
2. In the top-level project directory, run: *`cordova prepare`*

Loading Sync Definitions from Configuration Files (iOS Native Apps)

Loading methods are defined on the `MobileSyncSDKManager` class.

User store

Swift

```
MobileSyncSDKManager.sharedManager.setupUserSyncsFromDefaultConfig()
```

Objective-C

```
[[MobileSyncSDKManager sharedManager] setupUserSyncsFromDefaultConfig];
```

Global store

Swift

```
MobileSyncSDKManager.sharedManager.setupGlobalSyncsFromDefaultConfig()
```

Objective-C

```
[[MobileSyncSDKManager sharedManager] setupGlobalSyncsFromDefaultConfig];
```

Loading Sync Definitions from Configuration Files (Android Native Apps)

Loading methods are defined on the `MobileSyncSDKManager`. You can call these loaders from anywhere in your app. Make sure that the call occurs before you call any sync or resync functions.

User store

```
MobileSyncSDKManager.getInstance().setupUserSyncsFromDefaultConfig();
```

Global store

```
MobileSyncSDKManager.getInstance().setupGlobalSyncsFromDefaultConfig();
```

Using Sync Names

Mobile SDK provides a collection of APIs for using and managing named sync operations. You can programmatically create and delete named syncs at runtime, run or rerun them by name, and manage named syncs in memory.

Name-Based APIs (iOS)

Most of these methods are new. Updated methods use the same parameters as their existing analogs for `target`, `options`, and `updateBlock`.

Get sync status by name

Swift

```
var syncStatus = syncManager.getSyncStatus(syncName: syncState.syncName)
```

Objective-C

```
- (nullable
      SFSyncState*)getSyncStatusByName:(NSString*)syncName;
```

Check for an existing sync by name

Swift

```
var syncExists = syncManager.hasSync(syncName: syncState.syncName)
```

Objective-C

```
- (BOOL)hasSyncWithName:(NSString*)syncName;
```

Delete a sync configuration by name

Swift

```
syncManager.deleteSync(syncName: syncState.syncName)
```

Objective-C

```
- (void)deleteSyncByName:(NSString*)syncName;
```

Create, run, or rerun a named sync configuration

See Syncing Down and Syncing Up.

Call cleanResyncGhosts with a named sync configuration

See Calling cleanResyncGhosts Methods by Sync Name .

Name-Based APIs (Android)

Most of these methods are new. Overridden methods use the same parameters as their existing analogs for `target`, `options`, and `callback`.

Get sync status by name

```
public SyncState getSyncStatus(String name);
```

Check for an existing sync by name

```
public boolean hasSyncWithName(String name);
```

Delete a sync configuration by name

```
public void deleteSync(String name);
```

Create, run, or rerun a named sync configuration

See Syncing Down and Syncing Up.

Call cleanResyncGhosts with a named sync configuration

See Calling cleanResyncGhosts Methods by Sync Name.

Name-Based APIs (Hybrid)

Most of these methods are existing legacy APIs. Wherever a sync ID is accepted, you can pass the sync name instead.

Get sync status by name

You can use this function to determine if a sync configuration exists. It returns null if the sync configuration doesn't exist.

```
getSyncStatus(storeConfig, syncIdOrName, successCB, errorCB)
```

Delete a sync configuration by name

```
deleteSync(storeConfig, syncIdOrName, successCB, errorCB)
```

Create and run a named sync configuration

The legacy `syncDown()` function now includes a `syncName` parameter. If the name is provided, Mobile SDK creates a configuration with the given name. This function fails if the requested sync name already exists.

```
syncDown(storeConfig, target, soupName, options, syncName, successCB, errorCB)
```

```
syncUp(storeConfig, target, soupName, options, syncName, successCB, errorCB)
```

Run (or rerun) any sync by name

This existing method now has an overload that accepts either a sync ID or name.

```
reSync(storeConfig, syncIdOrName, successCB, errorCB)
```

Name-Based APIs (React Native)

Most of these methods are existing legacy APIs. Wherever a sync ID is accepted, you can pass the sync name instead.

Get sync status by name

This existing method now has an overload that accepts either a sync ID or a sync name.

```
getSyncStatus(storeConfig, syncIdOrName, successCB, errorCB)
```

Delete by name

This method, new in Mobile SDK 6.0, accepts either a sync ID or a sync name.

```
deleteSync(storeConfig, syncIdOrName, successCB, errorCB)
```

Create and run a sync with a name - new optional parameter syncName

This existing method now has an optional parameter that accepts a sync name.

```
syncDown(storeConfig, target, soupName, options, syncName, successCB, errorCB)
syncUp(storeConfig, target, soupName, options, syncName, successCB, errorCB)
```

Run (or rerun) any sync by name - overloaded to accept id or name

This existing method now has an overload that accepts either a sync ID or a sync name.

```
reSync(storeConfig, syncIdOrName, successCB, errorCB)
```

Example: **Invoking the Resync Method in Native iOS Apps**

Excerpt from `SObject.swift` from MobileSyncExplorerSwift template app. In this example, `updateRemoteData` calls `reSync` with a sync up configuration (`kSyncUpName`). If that operation succeeds, it then calls `refreshRemoteData` with a sync down configuration (`kSyncDownName`). This follow-up step ensures that the soup reflects all the latest changes on the server:

```
func updateRemoteData(_ onSuccess: @escaping ([SObjectData]) -> Void, onFailure:@escaping
 (NSError?, SyncState?) -> Void) -> Void {
    do {
        try self.syncMgr.reSync(named: kSyncUpName) { [weak self] (syncState) in
            guard let strongSelf = self else {
                return
            }
            switch (syncState.status) {
            case .done:
                do {
                    let objects = try strongSelf.queryLocalData()
                    strongSelf.populateDataRows(objects)
                    try strongSelf.refreshRemoteData({ (sobjs) in
                        onSuccess(sobjs)
                    }, onFailure:  { (error,syncState) in
                        onFailure(error,syncState)
                    }
                    )
                } catch let error as NSError {
                    MobileSyncLogger.e(SObjectDataManager.self, message: "Error with
Resync \(error)" )
                    onFailure(error,syncState)
                }
                break
            case .failed:
                MobileSyncLogger.e(SObjectDataManager.self, message: "Resync
\(syncState.syncName) failed" )
                onFailure(nil,syncState)
                break
            default:
                break
            }
        }
    } catch {
        onFailure(error as NSError, nil)
```

```
    }
}
```

```
func refreshRemoteData(_ completion: @escaping ([SObjectData]) -> Void,
                          onFailure: @escaping (NSError?, SyncState) -> Void)
                          throws -> Void {
    try self.syncMgr.reSync(named: kSyncDownName) { [weak self] (syncState) in
        switch (syncState.status) {
        case .done:
            do {
                let objects = try self?.queryLocalData()
                self?.populateDataRows(objects)
                completion(self?.fullDataRowList ?? [])
            } catch {
                MobileSyncLogger.e(SObjectDataManager.self, message: "Resync
\(syncState.syncName) failed \(error)" )
            }
             break
        case .failed:
             MobileSyncLogger.e(SObjectDataManager.self, message: "Resync
\(syncState.syncName) failed" )
             onFailure(nil,syncState)
        default:
            break
        }
    }
}
```

Example: **Invoking the Resync Method in Native Android Apps**

Excerpt from `SObjectSyncableRepoBase.kt` from MobileSyncExplorerKotlinTemplate.

```
private suspend fun doSyncUp(): SyncState {
    try {
        return syncManager.suspendReSync(syncUpName)
    } catch (es: SyncManager.ReSyncException.FailedToStart) {
        throw SyncUpException.FailedToStart(cause = es)
    } catch (ef: SyncManager.ReSyncException.FailedToFinish) {
        throw SyncUpException.FailedToFinish(cause = ef)
    }
}
```

Excerpt from `ContactListLoader.java` from Android MobileSyncExplorer native sample app:

```
public synchronized void syncUp() {
    try {
       syncMgr.reSync(SYNC_UP_NAME /* see usersyncs.json */, new SyncUpdateCallback()
 {
        @Override
            public void onUpdate(SyncState sync) {
                if (Status.DONE.equals(sync.getStatus())) {
                    syncDown();
                }
            }
        });
```

```
    } catch (JSONException e) {
        Log.e(TAG, "JSONException occurred while parsing", e);
    } catch (MobileSyncException e) {
        Log.e(TAG, "MobileSyncException occurred while attempting to sync up", e);
    }
}
```

```
public synchronized void syncDown() {
    try {
       syncMgr.reSync(SYNC_DOWN_NAME /* see usersyncs.json */, new SyncUpdateCallback()
 {
        @Override
            public void onUpdate(SyncState sync) {
                if (Status.DONE.equals(sync.getStatus())) {
                    fireLoadCompleteIntent();
                }
            }
        });
    } catch (JSONException e) {
        Log.e(TAG, "JSONException occurred while parsing", e);
    } catch (MobileSyncException e) {
        Log.e(TAG, "MobileSyncException occurred while attempting to sync down", e);
    }
}
```

Example: **Invoking the Resync Method in Hybrid Apps**

Excerpt from `MobileSyncExplorer.html` from MobileSyncExplorerHybrid sample app:

```
syncDown: function() {
    cordova.require("com.salesforce.plugin.mobilesync").reSync("syncDownContacts" /*
see usersyncs.json */,
                    this.handleSyncUpdate.bind(this));
},
syncUp: function() {
   cordova.require("com.salesforce.plugin.mobilesync").reSync("syncUpContacts" /* see
 usersyncs.json */,
                   this.handleSyncUpdate.bind(this));},
```

Syncing Data

In native Mobile Sync apps, you can use the sync manager to sync data easily between the device and the Salesforce server. The sync manager provides methods for syncing "up"—from the device to the server—or "down"—from the server to the device.

All data requests in Mobile Sync apps are asynchronous. Asynchronous means that the sync method that you call returns the server response in a callback method or update block that you define.

Each sync up or sync down method returns a sync state object. This object contains the following information:

- Sync operation ID. You can check the progress of the operation at any time by passing this ID to the sync manager's `getSyncStatus` method.
- Your sync parameters (soup name, target for sync down operations, options for sync up operations).
- Type of operation (up or down).
- Progress percentage (integer, 0–100).

- Total number of records in the transaction.

Using the Sync Manager

The sync manager object handles simple sync up and sync down operations. For sync down, it sends authenticated requests to the server on your behalf, and stores response data locally in SmartStore. For sync up, it collects the records you specify from SmartStore and merges them with corresponding server records according to your instructions. Sync managers know how to handle authentication for Salesforce users and community users. Sync managers can store records in any user or global SmartStore instance—the default instance, or a named instance.

Sync manager classes provide factory methods that return customized sync manager instances. To use the sync manager, you create an instance that matches the requirements of your sync operation. For example, Mobile SDK provides a specialized sync manager class for layouts and another for metadata.

Important: It is of utmost importance that you create the correct type of sync manager for every sync activity. If you don't, your customers can encounter runtime authentication failures.

Once you've created an instance, you can use it to call typical sync manager functionality:

- Sync down
- Sync up
- Resync
- Stop
- Restart

Sync managers can perform three types of actions on SmartStore soup entries and Salesforce records:

- Create
- Update
- Delete

If you provide custom targets, sync managers can use them to synchronize data at arbitrary REST endpoints.

Sync Manager States

At runtime, sync manager objects progress through three states:

- `accepting_syncs`—The sync manager can start sync operations.
- `stopping`—The sync manager's `stop` method has been called.
- `stopped`—All sync operations have stopped.

The sync manager can start sync operations only when it's in the `accepting_syncs` state. Calling the sync manager's `restart` method resets a `stopping` or `stopped` state to `accepting_syncs`. If the `restart` method is called with `restartStoppedSyncs` set to true, the sync manager calls `reSync` on each stopped sync operation.

Sync Manager Instantiation (iOS)

Swift Class	Objective-C Class
`SyncManager`	`SFMobileSyncSyncManager`

In iOS, you use pairs of access and removal methods. You call the `sharedInstance` class methods on the `SFMobileSyncSyncManager` class to access a preconfigured shared instance for each scenario. When you're finished using the shared instance for a particular use case, remove it with the corresponding `removeSharedInstance` method.

For a specified user:

Swift

```
SFMobileSyncSyncManager.sharedInstance(user: userAccount)
SFMobileSyncSyncManager.removeSharedInstance(user: userAccount)
```

Objective-C

```
+ (instancetype)sharedInstance:(SFUserAccount *)user;
+ (void)removeSharedInstance:(SFUserAccount *)user;
```

For a specified user using the specified SmartStore database:

Swift

```
SFMobileSyncSyncManager.sharedInstance(forUser: userAccount, storeName: "StoreName")
SFMobileSyncSyncManager.removeSharedInstance(forUser: userAccount, storeName:
"StoreName")
```

Objective-C

```
+ (instancetype)sharedInstanceForUser:(SFUserAccount *)user
                            storeName:(NSString *)storeName;

+ (void)removeSharedInstanceForUser:(SFUserAccount *)user
                          storeName:(NSString *)storeName;
```

For the current user and a specified SmartStore database:

Swift

```
SFMobileSyncSyncManager.sharedInstance(for: store)
SFMobileSyncSyncManager.removeSharedInstance(for: store)
```

Objective-C

```
+ (instancetype)sharedInstanceForStore:(SFSmartStore *)store;
+ (void)removeSharedInstanceForStore:(SFSmartStore *)store;
```

Sync Manager Instantiation (Android)

In Android, you use a different factory method for each of the following scenarios:

For the current user:

```
public static synchronized SyncManager getInstance();
```

For a specified user:

```
public static synchronized SyncManager
getInstance(UserAccount account);
```

For a specified user in a specified community:

```
public static synchronized SyncManager
getInstance(UserAccount account, String communityId);
```

For a specified user in a specified community using the specified SmartStore database:

```
public static synchronized SyncManager
getInstance(UserAccount account, String communityId, SmartStore smartStore);
```

Example: Here's a Swift example of initializing the sync manager with a shared store.

```
store = SFSmartStore.sharedStore(withName: kDefaultSmartStoreName) as? SFSmartStore
syncManager = SFMobileSyncSyncManager.sharedInstance(for:store!)
```

Using the Sync Manager with Global SmartStore

To use Mobile Sync with a global SmartStore instance, call a static factory method on the sync manager object to get a compatible sync manager instance.

iOS

Swift

```
SyncManager.sharedInstance(store: store!)
```

Objective-C

```
+ (instancetype)
    sharedInstanceForStore:
        (SFSmartStore *)store;
```

Returns a sync manager instance that talks to the server as the current user and writes to or reads from the given SmartStore instance. Use this factory method for syncing data with the global SmartStore instance.

Android

```
SyncManager getInstance(UserAccount account, String communityId, SmartStore smartStore);
```

Returns a sync manager instance that talks to the server as the given community user and writes to or reads from the given SmartStore instance. Use this factory method for syncing data with the global SmartStore instance.

Hybrid

In each of the following methods, the optional first argument tells the Mobile Sync plug-in whether to use a global store. This argument accepts a Boolean value or a `StoreConfig` object. If you use a `StoreConfig` object, you can specify `storeName`, `isGlobalStore`, or both, depending on your context. See Creating and Accessing User-based Stores.

- ```
  syncDown(isGlobalStore, target, soupName, options, successCB, errorCB);
  syncDown(storeConfig, target, soupName, options, successCB, errorCB);
  ```
- ```
  reSync(isGlobalStore, syncId, successCB, errorCB);
  reSync(storeConfig, syncId, successCB, errorCB);
  ```
- ```
  syncUp(isGlobalStore, target, soupName, options, successCB, errorCB);
  syncUp(storeConfig, target, soupName, options, successCB, errorCB);
  ```
- ```
  getSyncStatus(isGlobalStore, syncId, successCB, errorCB);
  getSyncStatus(storeConfig, syncId, successCB, errorCB);
  ```

SEE ALSO:

Creating and Accessing User-based Stores

Using Global SmartStore

About Sync Statuses

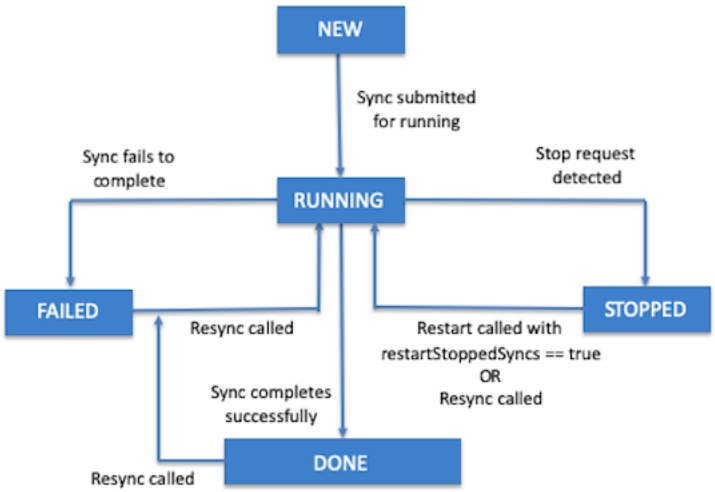
To enable full control over "in-flight" sync operations, Mobile SDK 7.1 adds a new status that indicates that the operation has been stopped.

Before Mobile SDK 7.1, each sync operation progressed through a cycle of statuses: `new`, `running`, and either `failed` or `done`. Mobile SDK 7.1 adds a `stopped` status to the final step, changing the cycle as follows:

1. `new`
2. `running`
3. `stopped` | `failed` | `done`

When key events occur, sync statuses are updated as follows.

Event	Previous Status	Reported Status
App creates a sync object	—	new
Sync object is submitted for running by a call to a sync down or sync up method, or by a first call to `reSync`	new	running
Sync operation completes successfully	running	done
Sync operation fails to complete	running	failed
A sync manager stop request is detected	running	stopped
Sync manager restarts with the `restartStoppedSyncs` parameter set to true	stopped	running
`reSync` is called	stopped/failed/done	running



Note:

- Any call to a `reSync` method on a currently running sync operation fails.
- Any call to a `reSync`, `syncDown`, `syncUp`, or `cleanResyncGhosts` method fails if the sync manager state is not in the `accepting_syncs` state.
- If any sync operations are running when the sync manager is first initialized—for example, if the app recently crashed shortly after starting syncs—the sync manager sets their statuses to `stopped`.

Getting Sync Objects by Status

Beginning with Mobile SDK 7.1, sync manager objects support a `getSyncsWithStatus` method that returns all sync state objects whose status matches a given value. You can use this method, for example, to iterate through all stopped syncs and restart them manually.

Note: To assume manual restart control over stopped syncs, call the sync manager's `restart` method with `restartStoppedSyncs` set to false.

iOS

Swift

```
open class func getSyncsWithStatus(_ store: SmartStore,
                                    status: SyncStatus) -> [SyncState]
```

Objective-C

```
+ (NSArray<SFSyncState>) getSyncsWithStatus:(SFSmartStore*)store
                                     status:(SFSyncStateStatus)status;
```

Android (Java)

```
public static List<SyncState> getSyncsWithStatus(SmartStore store, Status status)
```

Incremental Syncs with reSync

For sync up targets and SOQL-based sync down targets, you can incrementally update a pre-defined sync operation. Incremental `reSync` methods download or upload only new or updated records from the source. You can call `reSync` with either a sync ID or a sync name. If you call `reSync` for a sync configuration that has never been run, `reSync` knows to do a full sync.

The following general rules help you decide whether to use `syncUp` and `syncDown`, or `reSync`.

- You can use `syncUp` or `syncDown` anytime you want to do a full sync. You can pass in a sync you've defined previously, or you can define the sync in the call.
- You can use `reSync` only to run syncs that are already defined. On the first run, `reSync` performs a full sync. In subsequent syncs, `reSync` returns the delta between the last run and the current state.

During sync down, Mobile SDK checks downloaded records for the modification date field specified by the target and remembers the most recent timestamp. If you request a resync for that sync down, Mobile SDK passes the most recent timestamp, if available, to the sync down target. The sync down target then fetches only records created or updated since the given timestamp. The default modification date field is `lastModifiedDate`.

Of the three built-in sync down targets (MRU, SOSL-based, and SOQL-based), only the SOQL-based sync down target supports `reSync`. To support `reSync` in custom sync down targets, use the `maxTimeStamp` parameter passed during a fetch operation.

Limitation: There Are Ghosts in the Store!

After an incremental sync down, the following unused records remain in the local soup:

- Records that have been deleted on the server
- Records that no longer satisfy the sync down target

If you choose to remove these orphaned records, you can:

- Run a full sync down operation, or
- Call one of the `cleanResyncGhosts` methods

Invoking the reSync Method

iOS:

On a sync manager instance, call:

Swift

```
var syncState =
    try syncManager.reSync(id: syncState.syncId, onUpdate: updateFunc)
```

Objective-C

```
- (nullable SFSyncState*) reSync:(NSNumber*)syncId
updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;
```

Android:

On a `SyncManager` instance, call:

Kotlin

```
suspend fun suspendReSync(syncId: Long): SyncState
```

Java

```
SyncState reSync(long syncId, SyncUpdateCallback callback);
```

Hybrid:

Call:

```
cordova.require("com.salesforce.plugin.MobileSync").reSync(syncId,successCB);
```

Sample Apps

iOS

The MobileSyncExplorer sample app uses `reSync()` in the `SObjectDataManager` class.

Android

The MobileSyncExplorer sample app uses `reSync()` in the `ContactListLoader` class.

Hybrid

The SimpleSync sample app uses `reSync()` in `SimpleSync.html`'s `app.views.SearchPage` class.

Syncing Down

To download sObjects from the server to your local Mobile Sync soup, use the appropriate "sync down" method.

For sync down methods, you define a target that provides the list of sObjects to be downloaded. To provide an explicit list, use `JSONObject` on Android, or `NSDictionary` on iOS. However, you can also define the target with a query string. The sync target class provides factory methods for creating target objects from a SOQL, SOSL, or MRU query.

You also specify the name of the SmartStore soup that receives the downloaded data. This soup is required to have an indexed string field named `__local__`. Mobile SDK reports the progress of the sync operation through the callback method or update block that you provide.

Merge Modes

Sync down methods support an option that lets you control how incoming data merges with locally modified records. Choose one of the following behaviors:

1. Overwrite modified local records and lose all local changes. Set the `options` parameter to the following value:
   - **iOS:**

     Swift

     ```
     SyncOptions.newSyncOptions(forSyncDown: SyncMergeMode.overwrite)
     ```

     Objective-C

     ```
     [SFSyncOptions newSyncOptionsForSyncDown:SFSyncStateMergeModeOverwrite]
     ```

   - **Android:** `SyncOptions.optionsForSyncDown(MergeMode.OVERWRITE)`
2. Preserve all local changes and locally modified records. Set the `options` parameter to the following value:

- **iOS:**

  Swift

  ```
  SyncOptions.newSyncOptions(forSyncDown: SyncMergeMode.leaveIfChanged)
  ```

  Objective-C

  ```
  [SFSyncOptions newSyncOptionsForSyncDown:SFSyncStateMergeModeLeaveIfChanged]
  ```

- **Android:** `SyncOptions.optionsForSyncDown(MergeMode.LEAVE_IF_CHANGED)`

Important:

- If you use a version of `syncDown` that doesn't take an `options` parameter, existing sObjects in the cache can be overwritten. To preserve local changes, always run sync up before running sync down.
- Sync down payloads don't reflect records that have been deleted on the server. As a result, the update operation doesn't automatically delete the corresponding records in the target soups. These records that remain in the soup after deletion on the server are known as *ghosts*. To delete them, call one of the cleanResyncGhosts methods after you sync down.

iOS Sync Manager Methods

Swift Class Name

```
SyncManager
```

Objective-C Class Name

```
SFMobileSyncSyncManager
```

To create a sync down operation without running it:

Swift

```
var syncState = syncManager.createSyncDown(target: target, options: options,
    soupName: CONTACTS_SOUP, syncName: syncState.syncName)
```

Objective-C

```
- (SFSyncState *)createSyncDown:(SFSyncDownTarget *)target
                        options:(SFSyncOptions *)options
                       soupName:(NSString *)soupName
                       syncName:(NSString *)syncName;
```

To create and run a sync down operation that overwrites any local changes:

Swift

```
func syncDown(target: SFSyncDownTarget, soupName: String) -> SyncState
```

Objective-C

```
- (SFSyncState*) syncDownWithTarget:(SFSyncDownTarget*)target
                           soupName:(NSString*)soupName
                        updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;
```

To create and run an unnamed sync down operation:

Swift

```
var syncState = syncManager.syncDown(target: target, soupName: CONTACTS_SOUP,
onUpdate:updateFunc)
```

Objective-C

```
- (SFSyncState*) syncDownWithTarget:(SFSyncDownTarget*)target
                            options:(SFSyncOptions*)options
                           soupName:(NSString*)soupName
                        updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;
```

To create and run a named sync down operation:

Swift

```
var syncState = try syncManager.syncDown(target: target, options: options,
    soupName: CONTACTS_SOUP, syncName: syncState.syncName, onUpdate:updateFunc)
```

Objective-C

```
- (SFSyncState*) syncDownWithTarget:(SFSyncDownTarget*)target
                            options:(SFSyncOptions*)options
                           soupName:(NSString*)soupName
                           syncName:(NSString* __nullable)syncName
                        updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
                              error:(NSError**)error;
```

To run a named sync down operation:

Swift

```
var syncState =
    try syncManager.reSync(named: syncState.syncName, onUpdate: updateFunc)
```

Objective-C

```
- (nullable SFSyncState*) reSyncByName:(NSString*)syncName
                           updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
                                 error:(NSError**)error;
```

To rerun a previous sync operation using its sync ID:

Swift

```
var syncState =
    try syncManager.reSync(id: syncState.syncId, onUpdate: updateFunc)
```

Objective-C

```
- (nullable SFSyncState*) reSync:(NSNumber*)syncId
                     updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
                           error:(NSError**)error;
```

Android SyncManager Methods

To create a sync down operation without running it:

```
public SyncState createSyncDown(SyncDownTarget target,
    SyncOptions options, String soupName, String syncName)
    throws JSONException;
```

To create and run a sync down operation that overwrites any local changes:

```
public SyncState syncDown(SyncDownTarget target, String soupName,
    SyncUpdateCallback callback) throws JSONException;
```

To create and run an unnamed sync down operation:

```
public SyncState syncDown(SyncDownTarget target, SyncOptions options,
    String soupName, SyncUpdateCallback callback)
    throws JSONException;
```

To create and run a named sync down operation:

```
public SyncState syncDown(SyncDownTarget target, SyncOptions options,
    String soupName, String syncName, SyncUpdateCallback callback)
    throws JSONException;
```

To run or rerun a named sync configuration:

```
public SyncState reSync(String syncName, SyncUpdateCallback callback)
    throws JSONException;
```

To rerun a previous sync operation using its sync ID:

```
public SyncState reSync(long syncId, SyncUpdateCallback callback)
    throws JSONException;
```

Example: **iOS:**

The MobileSyncExplorerSwift sample app demonstrates how to use named syncs and sync configuration files with the Salesforce Contact object. In iOS, this sample defines a `ContactSObjectData` class that represents a contact record as a Swift object. The sample also defines several support classes:

- `ContactSObjectDataSpec`
- `SObjectData`
- `SObjectDataSpec`
- `SObjectDataFieldSpec`
- `SObjectDataManager`

To sync Contact data with the SmartStore soup, this app defines the following named sync operations in the `Resources/usersyncs.json` file:

```
{
  "syncs": [
    {
      "syncName": "syncDownContacts",
      "syncType": "syncDown",
      "soupName": "contacts",
      "target": {"type":"soql", "query":"SELECT FirstName, LastName, Title,
          MobilePhone, Email, Department, HomePhone FROM Contact LIMIT 10000"},
```

```
      "options": {"mergeMode":"OVERWRITE"}
    },
    {
      "syncName": "syncUpContacts",
      "syncType": "syncUp",
      "soupName": "contacts",
      "target": {"createFieldlist":["FirstName", "LastName", "Title",
          "MobilePhone", "Email", "Department", "HomePhone"]},
      "options": {"fieldlist":["Id", "FirstName", "LastName", "Title",
          "MobilePhone", "Email", "Department", "HomePhone"],
          "mergeMode":"LEAVE_IF_CHANGED"}
    }
  ]
}
```

In the `RootViewController` class, the `syncUpDown()` method starts the flow by calling the `updateRemoteData(_:onFailure:)` method of `SObjectDataManager`.

```
func syncUpDown(){
    let alert = self.showAlert("Syncing", message: "Syncing with Salesforce")
    sObjectsDataManager.updateRemoteData({ [weak self] (sObjectsData) in
        DispatchQueue.main.async {
            alert.message = "Sync Complete!"
            alert.dismiss(animated: true, completion: nil)
            self?.refreshList()
        }
    }, onFailure: { [weak self] (error, syncState) in
        alert.message = "Sync Failed!"
        alert.dismiss(animated: true, completion: nil)
        self?.refreshList()
    })
}
```

For the first argument of `updateRemoteData`, which represents success, `syncUpDown` passes a block that calls the `refreshList()` method of `RootViewController`. This method filters the local contacts according to customer input and refreshes the view.

`updateRemoteData` performs a sync up ensures that allowed soup changes are merged into the Salesforce org. If sync up succeeds—that is, if the `SyncState` status indicates "done"—then `updateRemoteData` does the following:

1. Retrieves all raw data from the freshly updated soup.
2. Transforms the soup's data to `ContactSObjectData` objects and stores these objects in an internal array.
3. Passes control to `refreshRemoteData(_:onFailure:)`. The `onSuccess` argument passed to `refreshRemoteData` is the block passed in from `syncUpDown`.

```
func updateRemoteData(_ onSuccess: @escaping ([SObjectData]) -> Void,
                      onFailure:@escaping (NSError?, SyncState?) -> Void) -> Void
 {
...
    try strongSelf.refreshRemoteData({ (sobjs) in
        onSuccess(sobjs)
    }, onFailure:  { (error,syncState) in
        onFailure(error,syncState)
```

```
    })
….
```

In `refreshRemoteData`, the app again calls `reSync` but with the `syncDownContacts` model—aliased as `kSyncDownName`—to update the soup. If sync down succeeds, `refreshRemoteData` "closes the circle" by executing the block that's passed to it from `syncUpDown` via `updateRemoteData`.

```
func refreshRemoteData(_ completion: @escaping ([SObjectData]) -> Void,onFailure:
@escaping (NSError?, SyncState) -> Void  ) throws -> Void {

    try self.syncMgr.reSync(named: kSyncDownName) { [weak self] (syncState) in
        switch (syncState.status) {
        case .done:
            do {
                let objects = try self?.queryLocalData()
                self?.populateDataRows(objects)
                completion(self?.fullDataRowList ?? [])
            } catch {
               MobileSyncLogger.e(SObjectDataManager.self,
                   message: "Resync \(syncState.syncName) failed \(error)" )
            }
            break
        case .failed:
             MobileSyncLogger.e(SObjectDataManager.self,
                   message: "Resync \(syncState.syncName) failed" )
             onFailure(nil,syncState)
        default:
            break
        }
    }
}
```

To summarize everything that happens in the `syncUpDown` call stack:

1. *Sync up*: It syncs soup changes up to the server by calling `updateRemoteData` on `SObjectsDataManager`. This step ensures that all allowable local and offline changes are merged into Salesforce.
2. *Sync down*: After the soup records are merged with server data, it syncs server data down to the soup through a call to `refreshRemoteData`. This step ensures that the soup reflects changes originating on the server and also changes merged from sync up. **Remember:** The sync up merge mode determines which soup edits are allowed on the server.
3. Finally, it updates its UI with the updated contact records from the soup.

Important: When you're syncing records, always call sync down after sync up as demonstrated by the MobileSyncExplorerSwift sample app.

Example: Android:

The native MobileSyncExplorer sample app demonstrates how to use Mobile Sync named syncs and sync configuration files with Contact records. In Android, it defines a `ContactObject` class that represents a Salesforce Contact record as a Java object. To sync Contact data down to the SmartStore soup, the `syncDown()` method resyncs a named sync down configuration that defines a SOQL query.

In the following snippet, the `reSync()` method loads the following named sync operations from the `res/raw/usersyncs.json` file:

```
{
  "syncs": [
    {
      "syncName": "syncDownContacts",
      "syncType": "syncDown",
      "soupName": "contacts",
      "target": {"type":"soql", "query":"SELECT FirstName, LastName, Title,
           MobilePhone, Email, Department, HomePhone FROM Contact LIMIT 10000",
           "maxBatchSize":500},
      "options": {"mergeMode":"OVERWRITE"}
    },
    {
      "syncName": "syncUpContacts",
      "syncType": "syncUp",
      "soupName": "contacts",
      "target": {"createFieldlist":["FirstName", "LastName", "Title", "MobilePhone",

          "Email", "Department", "HomePhone"]},
      "options": {"fieldlist":["Id", "FirstName", "LastName", "Title", "MobilePhone",

          "Email", "Department", "HomePhone"], "mergeMode":"LEAVE_IF_CHANGED"}
    }
  ]
}
```

If the sync down operation succeeds—that is, if `sync.getStatus()` equals `Status.DONE`—the received data goes into the specified soup. The callback method then fires an intent that reloads the data in the Contact list.

```
public synchronized void syncDown() {

    try {
  syncMgr.reSync(SYNC_DOWN_NAME /* see usersyncs.json */, new SyncUpdateCallback() {
            @Override
            public void onUpdate(SyncState sync) {
                if (Status.DONE.equals(sync.getStatus())) {
                    fireLoadCompleteIntent();
                }
            }
        });
    } catch (JSONException e) {
        Log.e(TAG, "JSONException occurred while parsing", e);
    } catch (MobileSyncException e) {
        Log.e(TAG, "MobileSyncException occurred while attempting to sync down", e);
    }

}
```

Syncing Up

To apply local changes on the server, use one of the "sync up" methods. These methods update the server with data from the given SmartStore soup. They look for created, updated, or deleted records in the soup, and then replicate those changes on the server. The

`options` argument specifies a list of fields to be updated. In Mobile SDK5.1 and later, you can override this field list by initializing the sync manager object with separate field lists for create and update operations. See Handling Field Lists in Create and Update Operations .

Locally created objects must include an "attributes" field that contains a "type" field that specifies the sObject type. For example, for an account named Acme, use: `{Id:"local_x", Name: Acme, attributes: {type:"Account"}}`.

iOS: SFMobileSyncSyncManager Methods

- You can create a named sync without running it.

  Swift

  ```
  var syncState = syncManager.createSyncUp(target: target, options: options,
      soupName: CONTACTS_SOUP, syncName: syncState.syncName)
  ```

  Objective-C

  ```
  - (SFSyncState *)createSyncUp:(SFSyncUpTarget *)target
                        options:(SFSyncOptions *)options
                       soupName:(NSString *)soupName
                       syncName:(NSString *)syncName;
  ```

- You can create and run a sync with just options that uses the default target.

  Swift

  ```
  var syncState = syncManager.syncUp(options: options, soupName: CONTACTS_SOUP,
      onUpdate: updateFunc)
  ```

  Objective-C

  ```
  - (SFSyncState*) syncUpWithOptions:(SFSyncOptions*)options
                            soupName:(NSString*)soupName
                         updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;
  ```

- You can create and run a sync based on a target that you configure in code.

  Swift

  ```
  var syncState = syncManager.syncUp(options: options, soupName: CONTACTS_SOUP,
      onUpdate: updateFunc)
  ```

  Objective-C

  ```
  - (SFSyncState*) syncUpWithTarget:(SFSyncUpTarget*)target
                            options:(SFSyncOptions*)options
                           soupName:(NSString*)soupName
                        updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;
  ```

- Or, you can create and run a named sync. If you load a sync with the same name from a configuration file, this sync overrides it.

  Swift

  ```
  var syncState =
      try syncManager.syncUp(target: target, options: options,
          soupName: CONTACTS_SOUP, syncName: syncState.syncName,
          onUpdate: updateFunc)
  ```

Objective-C

```
- (SFSyncState*) syncUpWithTarget:(SFSyncUpTarget*)target
                          options:(SFSyncOptions*)options
                        soupName:(NSString*)soupName
                        syncName:(NSString*)syncName
                     updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
                           error:(NSError**)error;
```

- To run or rerun an existing named sync configuration:

  Swift

  ```
  open func reSync(named syncName: String,
             onUpdate updateBlock: @escaping SyncUpdateBlock) throws -> SyncState
  ```

  Objective-C

  ```
  - (nullable SFSyncState*) reSyncByName:(NSString*)syncName
                             updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
                                   error:(NSError**)error;
  ```

- Or, to rerun a previous sync operation by sync ID:

  Swift

  ```
  open func reSync(id syncId: NSNumber,
        onUpdate updateBlock: @escaping SyncUpdateBlock) throws -> SyncState
  ```

  Objective-C

  ```
  - (nullable SFSyncState*) reSync:(NSNumber*)syncId
                       updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
                             error:(NSError**)error;
  ```

- To rerun a sync without getting progress updates, use this function from the Mobile Sync Swift extension:

  Swift

  ```
  public func reSyncWithoutUpdates(named syncName: String,
      _ completionBlock: @escaping (Result<SyncState, MobileSyncError>) -> Void)
  ```

  Objective-C

  (Not available.)

- To sync up by external ID, see Syncing Up by External ID.

Android: SyncManager Methods

- You can create a named sync up configuration without running it.

```
public SyncState createSyncUp(SyncUpTarget target,
    SyncOptions options,
    String soupName,
    String syncName)
    throws JSONException;
```

- You can create and run an unnamed sync up configuration:

```
public SyncState syncUp(SyncUpTarget target,
    SyncOptions options,
    String soupName,
    SyncUpdateCallback callback)
    throws JSONException;
```

- You can create and run a named sync up configuration:

```
public SyncState syncUp(SyncUpTarget target,
    SyncOptions options,
    String soupName,
    String syncName,
    SyncUpdateCallback callback)
    throws JSONException;
```

- To run or rerun an existing named sync configuration:

```
public SyncState reSync(String syncName, SyncUpdateCallback callback)
    throws JSONException;
```

- Or, to rerun a previous sync operation by sync ID:

```
public SyncState reSync(long syncId, SyncUpdateCallback callback)
    throws JSONException;
```

- To sync up by external ID, see Syncing Up by External ID.

Specifying Merge Modes

For sync up operations, you can specify a mergeMode option. You can choose one of the following behaviors:

1. Overwrite server records even if they've changed since they were synced down to that client. When you call the `syncUp` method:
   - **iOS:** Set the `options` parameter to

     Swift

     ```
     SyncOptions.newSyncOptions(forSyncUp: ["Name"], mergeMode: SyncMergeMode.overwrite)
     ```

     Objective-C

     ```
     [SFSyncOptions newSyncOptionsForSyncUp: ["Name"],
     mergeMode:SFSyncStateMergeModeOverwrite]
     ```

   - **Android:** Set the `options` parameter to `SSyncOptions.optionsForSyncUp(fieldlist, SyncState.MergeMode.OVERWRITE)`
   - **Hybrid:** Set the `syncOptions` parameter to `{mergeMode:"OVERWRITE"}`
2. If any server record has changed since it was synced down to that client, leave it in its current state. The corresponding client record also remains in its current state. When you call the `syncUp()` method:
   - **iOS:** Set the `options` parameter to

     Swift

     ```
     SyncOptions.newSyncOptions(forSyncUp: ["Name"], mergeMode:
     SyncMergeMode.leaveIfChanged)
     ```

Objective-C

```
[SFSyncOptions newSyncOptionsForSyncUp:fieldlist
mergeMode:SFSyncStateMergeModeLeaveIfChanged]
```

- **Android:** Set the `options` parameter to `SyncOptions.optionsForSyncUp(fieldlist, SyncState.MergeMode.LEAVE_IF_CHANGED)`
- **Hybrid:** Set the `syncOptions` parameter to `{mergeMode:"LEAVE_IF_CHANGED"}`

If your local record includes the target's modification date field, Mobile SDK detects changes by comparing it to the server record's matching field. The default modification date field is `lastModifiedDate`. If your local records do not include the modification date field, the `LEAVE_IF_CHANGED` sync up operation reverts to an overwrite sync up.

Important: The `LEAVE_IF_CHANGED` merge requires extra round trips to the server. More importantly, the status check and the record save operations happen in two successive calls. In rare cases, a record that is updated between these calls can be prematurely modified on the server.

Example: **iOS:**

The MobileSyncExplorerSwift sample app demonstrates how to use named syncs and sync configuration files with the Salesforce Contact object. In iOS, this sample defines a `ContactSObjectData` class that represents a contact record as a Swift object. The sample also defines several support classes:

- `ContactSObjectDataSpec`
- `SObjectData`
- `SObjectDataSpec`
- `SObjectDataFieldSpec`
- `SObjectDataManager`

To sync Contact data with the SmartStore soup, this app defines the following named sync operations in the `Resources/usersyncs.json` file:

```
{
  "syncs": [
    {
      "syncName": "syncDownContacts",
      "syncType": "syncDown",
      "soupName": "contacts",
      "target": {"type":"soql", "query":"SELECT FirstName, LastName, Title,
          MobilePhone, Email, Department, HomePhone FROM Contact LIMIT 10000"},
      "options": {"mergeMode":"OVERWRITE"}
    },
    {
      "syncName": "syncUpContacts",
      "syncType": "syncUp",
      "soupName": "contacts",
      "target": {"createFieldlist":["FirstName", "LastName", "Title",
          "MobilePhone", "Email", "Department", "HomePhone"]},
      "options": {"fieldlist":["Id", "FirstName", "LastName", "Title",
          "MobilePhone", "Email", "Department", "HomePhone"],
          "mergeMode":"LEAVE_IF_CHANGED"}
    }
  ]
}
```

In the `RootViewController` class, the `syncUpDown()` method starts the flow by calling the `updateRemoteData(_:onFailure:)` method of `SObjectDataManager`.

```
func syncUpDown(){
    let alert = self.showAlert("Syncing", message: "Syncing with Salesforce")
    sObjectsDataManager.updateRemoteData({ [weak self] (sObjectsData) in
        DispatchQueue.main.async {
            alert.message = "Sync Complete!"
            alert.dismiss(animated: true, completion: nil)
            self?.refreshList()
        }
    }, onFailure: { [weak self] (error, syncState) in
        alert.message = "Sync Failed!"
        alert.dismiss(animated: true, completion: nil)
        self?.refreshList()
    })
}
```

For the first argument of `updateRemoteData`, which represents success, `syncUpDown` passes a block that calls the `refreshList()` method of `RootViewController`. This method filters the local contacts according to customer input and refreshes the view.

`updateRemoteData` calls `reSync` using the `syncUpContacts` model—aliased here as `kSyncUpName`—. Syncing up ensures that allowed soup changes are merged into the Salesforce org.

```
func updateRemoteData(_ onSuccess: @escaping ([SObjectData]) -> Void,
                      onFailure:@escaping (NSError?, SyncState) -> Void) -> Void {
    do {
        try self.syncMgr.reSync(named: kSyncUpName) { [weak self] (syncState) in
            guard let strongSelf = self else {
                return
            }
            switch (syncState.status) {
            case .done:
                do {
                    let objects = try strongSelf.queryLocalData()
                    strongSelf.populateDataRows(objects)
                    try strongSelf.refreshRemoteData({ (sobjs) in
                        onSuccess(sobjs)
                    }, onFailure:  { (error,syncState) in
                        onFailure(error,syncState)
                    })
                } catch let error as NSError {
                    MobileSyncLogger.e(SObjectDataManager.self,
                        message: "Error with Resync \(error)" )
                    onFailure(error,syncState)
                }
                break
            case .failed:
                MobileSyncLogger.e(SObjectDataManager.self,
                    message: "Resync \(syncState.syncName) failed" )
                onFailure(nil,syncState)
                break
            default:
                break
```

```
            }
        }
    } catch {
        onFailure(error as NSError, nil)
    }
}
```

If sync up succeeds—that is, if the `SyncState` status indicates "done"—several things happen:

1. `queryLocalData` retrieves all raw data from the freshly updated soup.

   ```
   let objects = try strongSelf.queryLocalData()
   ```

2. `populateDataRows` transforms the soup's data to `ContactSObjectData` objects and stores these objects in an internal array.

   ```
   strongSelf.populateDataRows(objects)
   ```

3. Control passes to `refreshRemoteData(_:onFailure:)`. The `refreshRemoteData` method looks similar to `updateRemoteData` with two exceptions:
   - It performs a sync down instead of sync up.
   - If sync down succeeds, it "closes the circle" by executing the block that's been passed to it from `syncUpDown` via `updateRemoteData`.

To summarize everything that happens in the `syncUpDown` call stack:

1. *Sync up*: It syncs soup changes up to the server by calling `updateRemoteData` on `SObjectsDataManager`. This step ensures that all allowable local and offline changes are merged into Salesforce.
2. *Sync down*: After the soup records are merged with server data, it syncs server data down to the soup through a call to `refreshRemoteData`. This step ensures that the soup reflects changes originating on the server and also changes merged from sync up. **Remember:** The sync up merge mode determines which soup edits are allowed on the server.
3. Finally, it updates its UI with the updated contact records from the soup.

Important: When you're syncing records, always apply a sync up-sync down pair in the sequence demonstrated by the MobileSyncExplorerSwift sample app.

Example: **Android:**

To sync up to the server, you call `syncUp()` with the same arguments as `syncDown()`: list of fields, name of source SmartStore soup, and an update callback. The only coding difference is that you can format the list of affected fields as an instance of `SyncOptions` instead of `SyncTarget`. Here's the way it's handled in the MobileSyncExplorer sample:

```
public synchronized void syncUp() {

    final SyncUpTarget target = new SyncUpTarget();

    final SyncOptions options =
       SyncOptions.optionsForSyncUp(Arrays.asList(ContactObject.CONTACT_FIELDS_SYNC_UP),


            MergeMode.LEAVE_IF_CHANGED);
```

```
    try {

        syncMgr.syncUp(target, options, ContactListLoader.CONTACT_SOUP,
            new SyncUpdateCallback() {


                @Override

                public void onUpdate(SyncState sync) {

                    if (Status.DONE.equals(sync.getStatus())) {

                        syncDown();

                    }

                }

            });

    } catch (JSONException e) {

        Log.e(TAG, "JSONException occurred while parsing", e);

    } catch (MobileSyncException e) {

        Log.e(TAG, "MobileSyncException occurred while attempting to sync up", e);

    }

}
```

In the internal `SyncUpdateCallback` implementation, this example takes the extra step of calling `syncDown()` when sync up is done. This step guarantees that the SmartStore soup remains up-to-date with any recent changes made to Contacts on the server.

Syncing Up by External ID

Mobile SDK enhances its sync-up functionality by adding the ability to sync up by external ID. To use this feature, you specify an external ID field name in the sync up target definition. If a soup record is marked as locally created, updated, or deleted and has an external ID value, Mobile Sync syncs it up using `upsert` instead of `create`. If the record also has a valid `Id` value, however, Mobile Sync updates the indicated Salesforce record.

You can configure the external ID field name in Mobile Sync configuration files or in code. External IDs are supported for standard sync up targets and parent-child sync up targets.

Advantage of Using Upserts

Upserts are useful for avoiding record duplication. When your sync target supports an external ID field name and a locally created record has a value for that field, Mobile Sync upserts, rather than creates, the server record.

True upsert behavior allows the app to deal with network disconnection gracefully. If the network disconnects during a sync up, the app typically doesn't know whether its last request reached the server. In this case, the app's logical reaction when connectivity returns is to resync.

However, consider what happens if the server did in fact received your request and created the record. In this case, resyncing without using `upsert` creates a duplicate record. If you've configured external ID field names in the sync up target, the resync can run safely: upsert finds the recently created record and merely rewrites the record's fields with their existing values.

Parent-Child Sync Up Recommendation

In parent-child scenarios, you can specify the external ID field name for parent or children, or both. For a graceful offline experience, it's best to define the external ID field name in parent and children sync configurations.

Configuring an External ID Field Name Declaratively

Here are examples of specifying an external ID field name declaratively for a target in sync configuration files.

Standard Sync Up Target

```
{
  "syncName": "myExampleSyncUp",
  "syncType": "syncUp",
  "soupName": "MySoup",
    "target": {
                "idFieldName": "Id",
                "modificationDateFieldName": "LastModifiedDate",
                "externalIdFieldName": "TheExternalId"
   },
   "options": {
                "fieldlist": ["Name", "Description"],
                "mergeMode": "OVERWRITE"
   }
}
```

Parent-Child Sync Up Target

```
{
   "syncName":"parentChildrenSyncUp",
   "syncType":"syncUp",
   "soupName":"accounts",
   "target":{
      "iOSImpl":"SFParentChildrenSyncUpTarget",

"androidImpl":"com.salesforce.androidsdk.mobilesync.target.ParentChildrenSyncUpTarget",

      "parent":{
         "idFieldName":"IdX",
         "externalIdFieldName":"ExternalIdX",
         "sobjectType":"Account",
         "modificationDateFieldName":"LastModifiedDateX",
         "soupName":"accounts"
      },
      "createFieldlist":[
         "IdX",
```

```
            "Name",
            "Description"
        ],
        "updateFieldlist":[
            "Name",
            "Description"
        ],
        "children":{
            "parentIdFieldName":"AccountId",
            "idFieldName":"IdY",
            "externalIdFieldName":"ExternalIdY",
            "sobjectType":"Contact",
            "modificationDateFieldName":"LastModifiedDateY",
            "soupName":"contacts",
            "sobjectTypePlural":"Contacts"
        },
        "childrenCreateFieldlist":[
            "LastName",
            "AccountId"
        ],
        "childrenUpdateFieldlist":[
            "FirstName",
            "AccountId"
        ],
        "relationshipType":"MASTER_DETAIL"
    },
    "options":{
        "fieldlist":[

        ],
        "mergeMode":"LEAVE_IF_CHANGED"
    }
}
```

Extended Example

For a "one-stop" example of many types of sync configurations, see shared/example.usersyncs.json in the github.com/forcedotcom/SalesforceMobileSDK-Shared GitHub repo.

Configuring External ID Field Name Programmatically

iOS

For standard sync up targets:

Swift

Set the `externalIdFieldName` property in your `SyncUpTarget` or `BatchSyncUpTarget` class.

Objective-C

Set the `externalIdFieldName` property in your `SFSyncUpTarget` or `SFBatchSyncUpTarget` class.

For parent-child sync up targets:

Swift

Objective-C

Pass a non-nil value to the `externalIdFieldName` parameters of these factory methods (new in Mobile SDK 9.0):

```
// From SFParentInfo.h
+ (SFParentInfo *)newWithSObjectType:(NSString *)sobjectType
                            soupName:(NSString *)soupName
                         idFieldName:(NSString *)idFieldName
           modificationDateFieldName:(NSString *)modificationDateFieldName
                 externalIdFieldName:(NSString * __nullable) externalIdFieldName;

// From SFChildrenInfo.h
+ (SFChildrenInfo *)newWithSObjectType:(NSString *)sobjectType
                      sobjectTypePlural:(NSString *)sobjectTypePlural
                               soupName:(NSString *)soupName
                      parentIdFieldName:(NSString *)parentIdFieldName
                            idFieldName:(NSString *)idFieldName
             modificationDateFieldName:(NSString *)modificationDateFieldName
                   externalIdFieldName:(NSString * __nullable)externalIdFieldName;
```

Android

For standard sync up targets:

Create your target with one of these constructors:

```
public SyncUpTarget(List<String> createFieldlist, List<String> updateFieldlist,
    String idFieldName, String modificationDateFieldName,
    String externalIdFieldName)
```

```
public BatchSyncUpTarget(List<String> createFieldlist, List<String> updateFieldlist,

    String idFieldName, String modificationDateFieldName,
    String externalIdFieldName)
```

For parent-child sync up targets:

Pass a non-nil value to the `externalIdFieldName` parameters of these factory methods (new in Mobile SDK 9.0):

```
// From ParentInfo.java
public ParentInfo(String sobjectType, String soupName, String idFieldName,
    String modificationDateFieldName, String externalIdFieldName)
```

```
// From ChildrenInfo.java
public ChildrenInfo(String sobjectType, String sobjectTypePlural, String soupName,
    String parentIdFieldName, String idFieldName,
    String modificationDateFieldName, String externalIdFieldName)
```

Stopping and Restarting Sync Operations

Beginning in Mobile SDK 7.1, sync manager classes provide methods that allow apps to stop and restart syncs. Sometimes, a stopped sync is only paused and can later be restarted. In other cases, a stop request forces the sync operation to fail with no possibility of resuming. These new APIs support another Mobile SDK 7.1 enhancement: The ability to share data safely across multiple apps, or between an app and its extensions.

Stop/restart APIs provide a crucial safeguard for sharing data across multiple apps, or between an app and its extensions. To prevent multiple processes from writing to the same store, a data-sharing app is required to pause sync operations when it moves to the background.

Important: Mobile SDK does NOT automatically call `stop` or `restart` when an app moves to the background or the foreground.

Stop/Restart Method Descriptions

Sync manager stop/restart methods control the following sync operations:

- Sync up
- Sync down
- Resync
- Clean resync ghosts

The following generic method descriptions apply to all supported platforms.

stop
Asks the sync manager to suspend all sync operations currently running or queued for running. Sync down, sync up, and clean resync ghosts tasks now check periodically for stop requests. When a sync down or sync up tasks detect a stop request, they immediately stop running and change their status to stopped. When a clean resync ghosts task detects a stop request, it stops running and reports a failure. Tasks submitted after the stop request immediately return an error.

restart

- Restarts the sync manager. A stopped sync manager can accept sync tasks again when `restart` is called. The `restart` method takes a Boolean argument (`restartStoppedSyncs`) and a callback block, and behaves as follows.
  - If `restartStoppedSyncs` is true, `restart` calls `reSync` on all stopped sync up and sync down tasks and sends updates to the callback block.
  - If `restartStoppedSyncs` is false, the tasks remain stopped. The app itself can then restart sync up and sync down operations as needed by calling `reSync`.

isStopping
Returns true if a stop was requested but some tasks are still running.

isStopped
Returns true if a stop was requested and all tasks have stopped running.

iOS

Stop/restart functionality is available only for native iOS platforms. Here are the platform-specific signatures.

Swift Class	Objective-C Class
`SyncManager`	`SFMobileSyncSyncManager`

Swift

```
open func stop()
open func restart(restartStoppedSyncs: Bool,
                  onUpdate updateBlock: @escaping SyncUpdateBlock) throws
```

```
open func isStopping() -> Bool
open func isStopped() -> Bool
```

Objective-C

```
- (void)stop;
- (BOOL)restart:(BOOL)restartStoppedSyncs
    updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
          error:(NSError**)error;
- (BOOL)isStopping;
- (BOOL)isStopped;
```

Android

```
public synchronized void stop();
public synchronized void restart(boolean restartStoppedSyncs,
    SyncUpdateCallback callback);
public boolean isStopping();
public boolean isStopped();
```

What About Hybrid and React Native?

Currently, `stop` and `restart` methods are not exposed through either the Cordova or the React Native bridge. These new APIs are most useful for responding to application life cycle events, such as moving to the background or foreground. Typically, apps handle this type of low-level behavior in native code.

Efficiently Restarting a Paused Sync Down Operation

Mobile Sync records a max time stamp. This setting reflects the maximum value of the field that contains the "last modified" date for downloaded records. Typically, Mobile SDK captures this value when a sync operation has completed. Before Mobile SDK 7.1, Mobile Sync used this value to reduce the number of records fetched on repeated `reSync` calls.

In Mobile SDK 7.1, sync down target base classes—`SyncDownTarget` on Android, `SFSyncDownTarget` on iOS—support a new method, `isSyncDownSortedByLatestModification()`, that subclasses can implement. This method returns a Boolean value that, when true, instructs Mobile Sync to sort a batch of records. When a sync down operation sorts records, the max time stamp is updated on the sync state object with every batch of records being fetched. If the sync down is stopped and later restarted, the max time stamp keeps previous batches from being refetched.

For SOQL-based sync down operations—`SoqlSyncDownTarget` on Android, `SFSoqlSyncDownTarget` on iOS—the sorting behavior depends on the given SOQL query. If the query does not include an `ORDER BY` clause, Mobile SDK adds an `ORDER BY last-modified-field` clause, and `isSyncDownSortedByLatestModification()` returns true.

Behavior also differs slightly between resyncing a sync down operation that has completed versus one that was stopped:

- When `reSync` is invoked for a completed sync down, Mobile SDK fetches only records whose time stamp exceeds the captured max time stamp.
- When `reSync` is invoked for a stopped sync down, Mobile SDK fetches records whose time stamp is greater than or equal to the captured max time stamp. This variation is necessary because records with the same time stamp could span more than one batch.

About Sync Task Errors

In Mobile SDK 7.1 and later, sync task methods return error information consistently across platforms.

Sync tasks—sync down, sync up, resync, and clean resync ghosts—can fail for the following reasons:

- Invalid ID or name
- The requested sync operation is already running
- Sync manager is stopped or stopping

To inform apps of the nature of failures, sync methods in Mobile SDK 7.1 and later return error information as follows.

- On iOS:
  - Swift methods throw a `SyncState` object.
  - Objective-C methods include an `NSError**` parameter.

  Earlier versions of these methods are deprecated.
- On Android, these methods throw `MobileSyncException`.

New and Updated Sync and Resync Methods in Mobile SDK 7.1

New iOS Methods

Objective-C

The following methods now support and `NSError` parameter.

```
- (nullable SFSyncState*)
    syncDownWithTarget:(SFSyncDownTarget*)target
               options:(SFSyncOptions*)options
              soupName:(NSString*)soupName
              syncName:(*nullable* NSString*)syncName
           updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
                 error:(NSError**)error
;
- (*nullable* SFSyncState*)
    syncUpWithTarget:(SFSyncUpTarget*)target
             options:(SFSyncOptions*)options
            soupName:(NSString*)soupName
            syncName:(*nullable* NSString*)syncName
         updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
               error:(NSError**)error;

- (*nullable* SFSyncState*)
         reSync:(NSNumber*)syncId
    updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
          error:(NSError**) error;

- (*nullable* SFSyncState*)
    reSyncByName:(NSString*)syncName
     updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
           error:(NSError**)error NS_SWIFT_NAME(reSync(named:onUpdate:));
```

Updated iOS Methods

Swift

The following methods now throw a `SyncState` object.

```
open func syncDown(target: SyncDownTarget,
                   options: SyncOptions,
                  soupName: String,
                  syncName: String?,
```

```
     onUpdate updateBlock: @escaping SyncUpdateBlock) throws -> SyncState

open func syncUp(target: SyncUpTarget,
                 options: SyncOptions,
                soupName: String,
                syncName: String?,
   onUpdate updateBlock: @escaping SyncUpdateBlock) throws -> SyncState

open func reSync(id syncId: NSNumber,
      onUpdate updateBlock: @escaping SyncUpdateBlock) throws -> SyncState

open func reSync(
          named syncName: String,
    onUpdate updateBlock: @escaping SyncUpdateBlock) throws -> SyncState
```

Deprecated iOS Methods in Mobile SDK 7.1

Objective-C

Existing sync methods that do not support an `NSError` output parameter are slated for removal in a future major release.

```
- (nullable SFSyncState*)
    syncDownWithTarget:(SFSyncDownTarget*)target
               options:(SFSyncOptions*)options
              soupName:(NSString*)soupName
              syncName:(*nullable* NSString*)syncName
         updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;

- (nullable SFSyncState*)
    syncUpWithTarget:(SFSyncUpTarget*)target
             options:(SFSyncOptions*)options
            soupName:(NSString*)soupName
            syncName:(*nullable* NSString*)syncName
        updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;

- (nullable SFSyncState*)
       reSync:(NSNumber*)syncId
    updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;

- (nullable SFSyncState*)
    reSyncByName:(NSString*)syncName
     updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;
```

See Also

For changes to `cleanResyncGhosts` methods, see Handling "Ghost" Records After Sync Down Operations.

Handling "Ghost" Records After Sync Down Operations

If you're finding that sync down operations sometimes leave unwanted records in your SmartStore soups, you can use the `cleanResyncGhosts` API to get rid of them.

In certain prescribed cases, SmartStore soups do not reflect the exact contents of the most recent sync down request. For example, if a record is deleted on the Salesforce server, the next sync down operation doesn't remove that record from SmartStore. Also, records that

don't satisfy the sync criteria are excluded from the sync down results but aren't automatically removed from the soup. These records that unexpectedly remain in the SmartStore soup are known as *ghosts*.

To root out these haunts, Mobile Sync provides a set of `cleanResyncGhosts` methods that identify and remove ghosts. You pass in the ID or name of a sync object and define a callback block. These methods are available for Android native, iOS native, hybrid, and React Native platforms.

Warning: Exercise restraint in using the `cleanResyncGhosts` methods! Calls to these methods can be expensive in both runtime performance and payload size. Use these methods for low-frequency cleanup, rather than as part of every sync down operation. Use your own judgment to determine whether a particular set of ghosts is problematic and therefore requires immediate cleanup.

Using cleanResyncGhosts with Custom Sync Down Targets

If your app uses a custom sync down target, `cleanResyncGhosts` requires the custom target to implement the `getListOfRemoteIds` method. This method returns the list of Salesforce IDs that satisfy the sync down target's criteria. For `getListOfRemoteIds` coding examples, see the SOQL, SOSL, or MRU sync down target in these Mobile Sync library folders:

iOS
https://github.com/forcedotcom/SalesforceMobileSDK-iOS/tree/master/libs/MobileSync/MobileSync/Classes/Util

Android
https://github.com/forcedotcom/SalesforceMobileSDK-Android/tree/master/libs/MobileSync/src/com/salesforce/androidsdk/mobilesync/util

Preparing Soups for cleanResyncGhosts

For the target soup, add an index for the following field:

`__sync_id__`
This field ensures that the `cleanResyncGhosts()` method removes only the desired soup elements. Mobile Sync manages the content of this field for you.

Calling cleanResyncGhosts Methods by Sync ID

iOS Native

Swift	Objective-C
`SyncManager`	`SFMobileSyncSyncManager`

Swift
In Mobile SDK 7.1, the following method is updated to throw.

```
open func cleanResyncGhosts(
                        forId syncId: NSNumber,
    onComplete completionStatusBlock: @escaping SyncCompletionBlock) throws
```

Objective-C
In Mobile SDK 7.1, the following method is updated to support an `NSError` output parameter.

```
- (BOOL)
        cleanResyncGhosts:(NSNumber*)syncId
    completionStatusBlock:
```

```
        (SFSyncSyncManagerCompletionStatusBlock)completionStatusBlock
                    error:(NSError**)error;
```

Android Native

Kotlin

```
suspend fun suspendCleanResyncGhosts(syncId: Long): Int
```

Java

```
public void cleanResyncGhosts(final long syncId)
    throws JSONException, IOException

public void cleanResyncGhosts(long syncId,
    final CleanResyncGhostsCallback callback)
    throws JSONException
```

Hybrid

```
cleanResyncGhosts(isGlobalStore, syncId, successCB, errorCB)

cleanResyncGhosts(storeConfig, syncId, successCB, errorCB)
```

React Native

```
mobilesync.cleanResyncGhosts(isGlobalStore, syncId, successCB, errorCB)

mobilesync.cleanResyncGhosts(storeConfig, syncId, successCB, errorCB)
```

Calling cleanResyncGhosts Methods by Sync Name

You can also call `cleanResyncGhosts` with a sync name.

iOS (Swift)

```
//Mobile Sync native Swift extension function
public func cleanGhosts(named syncName: String,
    _ completionBlock: @escaping (Result<UInt, MobileSyncError>) -> Void

//Objective-C function renamed for Swift
open func cleanResyncGhosts(
                    forName syncName: String,
    onComplete completionStatusBlock: @escaping SyncCompletionBlock) throws

//iOS 13 or above only:
//Mobile Sync native Swift extension function, using Combine Publisher
public func cleanGhostsPublisher(for syncName: String) ->
    Future<UInt, MobileSyncError>
```

iOS (Objective-C)

```
- (*BOOL*)
    cleanResyncGhostsByName:(NSString*)syncName
      completionStatusBlock:
          (SFSyncSyncManagerCompletionStatusBlock)completionStatusBlock
                    error:(NSError**)error;
```

```
- (*BOOL*)
        cleanResyncGhosts:(NSNumber*)syncId
    completionStatusBlock:
        (SFSyncSyncManagerCompletionStatusBlock)completionStatusBlock
                    error:(NSError**)error;
```

Android (Kotlin)

```
suspend fun suspendCleanResyncGhosts(syncId: Long): Int
```

Android (Java)

```
public void cleanResyncGhosts(final String syncName, final CleanResyncGhostsCallback
callback)
```

Deprecations in Mobile SDK 7.1

Deprecated iOS Method (Objective-C)

The following method that does not support an `NSError` output parameter is slated for removal in a future major release.

```
- (*BOOL*)
        cleanResyncGhosts:(NSNumber*)syncId
    completionStatusBlock:
        (SFSyncSyncManagerCompletionStatusBlock)completionStatusBlock;
```

Using Standard Targets

Mobile Sync provides ready-to-use target classes for several standard Salesforce request types. You can use these targets implicitly through configuration files, or directly through code.

Supported standard types include:

- Layout
- Metadata
- MRU
- SOQL query
- SOSL query
- Refresh
- Batch
- Briefcase
- sObject Collections

Using the MRU Sync Down Target

To retrieve the most recently viewed records for a specific Salesforce object in your org, use the MRU sync down target.

The MRU sync down target returns only the most recently viewed records for the given object, as determined by the Salesforce API.

Configuration File Usage

For the `"target"` property, specify the following values.

Target Properties

"type":"mru"

"fieldList": Array of <string>
List of fields to sync.

"objectType": <string>
Name of a Salesforce object.

iOS APIs

Swift

Class: `MruSyncDownTarget`

```
MruSyncDownTarget.newSyncTarget(objectType: String, fieldlist: [Any])
```

Objective-C

Class: `SFMruSyncDownTarget`

```
+ (SFMruSyncDownTarget*) newSyncTarget:(NSString*)objectType
fieldlist:(NSArray*)fieldlist;
```

Android APIs

Kotlin

Class: `MruSyncDownTarget`

```
public MruSyncDownTarget(fieldlist: List<String>, objectType: String)
```

Java

Class: `MruSyncDownTarget`

```
public MruSyncDownTarget(List<String> fieldlist, String objectType)
```

Example:

```
{
  "syncs": [
    {
      "syncName": "syncDownMruContacts",
      "syncType": "syncDown",
      "soupName": "contacts",
      "target": {"type":"mru",
                 "fieldlist":["FirstName", "LastName", "Title",
                              "MobilePhone", "Email",
                              "Department", "HomePhone"],
                              "object":"Contact"},
      "options": {"mergeMode":"OVERWRITE"}
    },
    {
      "syncName": "syncUpContacts",
      "syncType": "syncUp",
      "soupName": "contacts",
      "target": {"createFieldlist":["FirstName", "LastName",
                                    "Title", "MobilePhone",
```

```
                                    "Email", "Department",
                                    "HomePhone"]},
      "options": {"fieldlist":["Id", "FirstName", "LastName",
                                     "Title", "MobilePhone",
                                     "Email", "Department",
                                     "HomePhone"],
                  "mergeMode":"LEAVE_IF_CHANGED"}
    }
  ]
}
```

Using the SOQL Sync Down Target

If you can define a SOQL query that selects everything required for a business need, the SOQL target is your simplest sync down option. This target takes a SOQL query and optional supporting arguments.

Mobile Sync wraps the SOQL query you provide as a REST request and sends it to Salesforce.

Configuration File Usage

For the `"target"` property, specify the following values.

Target Properties

"type":"soql"

"query": <string>
The SOQL query.

"idFieldName": <string>
(Optional) Name of a custom ID field. If you provide `"idFieldName"`, Mobile Sync uses the field with the given name to get the ID of the record. For example, if you specify `"idFieldName":"AcmeId"`, Mobile Sync obtains the record's ID from the `AcmeId` field instead of the default `Id` field.

"modificationDateFieldName": <string>
(Optional) Name of the field containing the last modification date for the record. If you provide `modificationDateFieldName`, Mobile Sync uses the field with this name to compute the `maxTimestamp` value that `startFetch` uses to resync the records. Default field name is `lastModifiedDate`.

"maxBatchSize": <integer>
(Optional) Proposed number of records to obtain in each fetch operation. If you provide a `maxBatchSize` value, Mobile Sync uses it to suggest the maximum number of records to be returned by each fetch operation. The actual number of records fetched can be more or less than the given value. Actual runtime batch sizes can depend on performance concerns, number of matching records, or a LIMIT specified in the query.

iOS APIs

Swift

Class: `SoqlSyncDownTarget`

```
SoqlSyncDownTarget.newSyncTarget(_ query:String) → Self
```

```
SoqlSyncDownTarget.newSyncTarget(_ query:String, maxBatchSize size:Int) → Self
```

Objective-C

Class: `SFSoqlSyncDownTarget`

```
+ (SFSoqlSyncDownTarget*) newSyncTarget:(NSString*)query;
+ (SFSoqlSyncDownTarget*) newSyncTarget:(NSString*)query
    maxBatchSize:(NSInteger) maxBatchSize;
```

Android APIs

Kotlin

Class: `SoqlSyncDownTarget`

```
public fun SoqlSyncDownTarget(query: String)
public fun SoqlSyncDownTarget(idFieldName: String, modificationDateFieldName: String,
    query: String)
public fun SoqlSyncDownTarget(idFieldName: String, modificationDateFieldName: String,
    query: String, maxBatchSize: int)
```

Java

Class: `SoqlSyncDownTarget`

```
public SoqlSyncDownTarget(String idFieldName, String modificationDateFieldName,
    String query)
public SoqlSyncDownTarget(String idFieldName, String modificationDateFieldName,
    String query, int maxBatchSize)
```

Example:

```
{
  "syncs": [
    {
      "syncName": "syncDownContacts",
      "syncType": "syncDown",
      "soupName": "contacts",
      "target": {"type":"soql", "query":"SELECT FirstName, LastName, Title,
           MobilePhone, Email, Department, HomePhone FROM Contact LIMIT 10000",
           "maxBatchSize":500},
      "options": {"mergeMode":"OVERWRITE"}
    },
    {
      "syncName": "syncUpContacts",
      "syncType": "syncUp",
      "soupName": "contacts",
      "target": {"createFieldlist":["FirstName", "LastName", "Title", "MobilePhone",

          "Email", "Department", "HomePhone"]},
      "options": {"fieldlist":["Id", "FirstName", "LastName", "Title", "MobilePhone",

          "Email", "Department", "HomePhone"], "mergeMode":"LEAVE_IF_CHANGED"}
    }
  ]
}
```

Using the SOSL Sync Down Target

Mobile Sync wraps the SOSL query you provide as a REST request and sends it to Salesforce.

Configuration File Usage

For the `"target"` property, specify the following values.

Target Properties

"type":"sosl"

"query": <string>
The SOSL query.

"idFieldName": <string>
(Optional) Name of a custom ID field. If you provide `"idFieldName"`, Mobile Sync uses the field with the given name to get the ID of the record. For example, if you specify `"idFieldName":"AcmeId"`, Mobile Sync obtains the record's ID from the `AcmeId` field instead of the default `Id` field.

"modificationDateFieldName": <string>
(Optional) Name of the field containing the last modification date for the record. If you provide `modificationDateFieldName`, Mobile Sync uses the field with this name to compute the `maxTimestamp` value that `startFetch` uses to resync the records. Default field name is `lastModifiedDate`.

iOS APIs

These factory methods create a SOSL sync down target that defines the `"query"` property. To specify the optional `"idFieldName"` and `"modificationDateFieldName"` properties, set their superclass members on the returned target.

Swift
Class: `SoslSyncDownTarget`

```
SoslSyncDownTarget.newSyncTarget(_ query:String) → Self
```

Objective-C
Class: `SFSoslSyncDownTarget`

```
+ (SFSoslSyncDownTarget*) newSyncTarget:(NSString*)query;
```

Android APIs

These factory methods create a SOSL sync down target that contains the `"query"` property. To specify the optional `"idFieldName"` and `"modificationDateFieldName"` properties, set their superclass members on the returned target.

Kotlin
Class: `SoslSyncDownTarget`

```
public fun SoslSyncDownTarget(query: String)
```

Java
Class: `SoslSyncDownTarget`

```
public SoslSyncDownTarget(String query)
```

Example:

```
{
  "syncs": [
    {
      "syncName": "syncDownAcme",
      "syncType": "syncDown",
      "soupName": "contacts",
      "target": {"type":"soql", "query":"FIND 'Acme' IN ALL FIELDS RETURNING
Account(Name),
          Contact(FirstName,LastName)"},
      "options": {"mergeMode":"OVERWRITE"}
    }
  ]
}
```

Using the Briefcase Sync Down Target

If your org uses Briefcases for your mobile users, the Briefcase sync down target was introduced in Mobile SDK 10.1. This sync target is an efficient way to load many records at a time. Sync is constrained to the records included in all Briefcases assigned to the current user and made accessible in the mobile client's Connected App. You can limit the sync target to specific objects and fields included in those Briefcases. This target takes an array of `BriefcaseObjectInfo` objects, which include sObject type, fields, and the soup to add them to.

Tip: SOQL sync down targets can only get one type of record at a time, and Parent Children sync down targets support only getting parent records with their children. In contrast, Briefcase sync down targets are more flexible. Briefcases can be defined to include many different kinds of objects, including related objects up to three levels deep. Your sync target definition can include any or all of these objects.

A *briefcase* is a set of queries that together select a cohesive collection of related records, optimized for the current user. Briefcases are defined in advance using Briefcase Builder, but execution of briefcase queries is delayed until the briefcase is accessed by a client application. Briefcases can be used to select records for data priming in advance of going offline, and for other general data loading purposes.

Briefcases are assigned to specific users, and queries can include the current user in their criteria. Briefcases are made accessible to your mobile app via the mobile client's Connected App. The objects, records, and fields available via this sync target are automatically constrained by these configuration details.

Additionally, because briefcases are defined and managed by your org's admin, your admin can change the sync behavior of client apps without requiring any code changes.

For briefcase sync targets, Mobile Sync uses Salesforce APIs that are optimized for loading large numbers of records in a single session. If you need to load hundreds or thousands of records at a time, especially in preparation for going offline, this target can be more efficient than other options.

Configuration File Usage

For the `"target"` property, specify the following values.

Target Properties

"type":"briefcase"

"infos": array of <BriefcaseObjectInfo> items
An array of objects that describe the specific sObjects and fields to retrieve, along with the soup to place them in.

BriefcaseObjectInfo Properties

"soupName": <string>
Name of the soup to store records of this object type into during sync.

"sobjectType": <string>
Name of a Salesforce object to sync.

"fieldlist": array of <string>
List of fields to sync for this object.

"idFieldName": <string>
(Optional) Name of a custom ID field. If you provide `"idFieldName"`, Mobile Sync uses the field with the given name to get the ID of the record. For example, if you specify `"idFieldName":"AcmeId"`, Mobile Sync obtains the record's ID from the `AcmeId` field instead of the default `Id` field.

"modificationDateFieldName": <string>
(Optional) Name of the field containing the last modification date for the record. If you provide `modificationDateFieldName`, Mobile Sync uses the field with this name to compute the `maxTimestamp` value that `startFetch` uses to resync the records. Default field name is `lastModifiedDate`.

Required: "soupName", "sobjectType", "fieldlist"

iOS APIs

Create `BriefcaseObjectInfo` objects as needed, and then use them to create a sync down target object.

Swift

Class: `BriefcaseSyncDownTarget`

```
let briefcaseAccountInfo = BriefcaseObjectInfo(
    soupName: "soup_for_accounts",
    sobjectType: "Account",
    fieldlist: ["Name", "Description"])
let briefcaseContactInfo = BriefcaseObjectInfo(
    soupName: "soup_for_contacts",
    sobjectType: "Contact",
    fieldlist: ["LastName"])
let target = BriefcaseSyncDownTarget(
    infos: [briefcaseAccountInfo, briefcaseContactInfo])
```

Objective-C

Class: `SFBriefcaseSyncDownTarget`

```
SFBriefcaseObjectInfo *briefcaseAccountInfo = [[SFBriefcaseObjectInfo alloc]
    initWithSoupName:@"soup_for_accounts"
         sobjectType:@"Account"
           fieldlist: @[@"Name", @"Description"]];
SFBriefcaseObjectInfo *briefcaseContactInfo = [[SFBriefcaseObjectInfo alloc]
    initWithSoupName:@"soup_for_contacts"
         sobjectType:@"Contact"
           fieldlist: @[@"LastName"]];
SFBriefcaseSyncDownTarget *target = [[SFBriefcaseSyncDownTarget alloc]
    initWithInfos:@[briefcaseAccountInfo, briefcaseContactInfo]];
```

Android APIs

Create `BriefcaseObjectInfo` objects as needed, and then use them to create a sync down target object.

Java

Class: `BriefcaseSyncDownTarget`

```
BriefcaseSyncDownTarget target = new BriefcaseSyncDownTarget(
    Arrays.asList(
        new BriefcaseObjectInfo(
            "soupForAccounts",
            "Account",
            Arrays.asList("Name", "Description")),
        new BriefcaseObjectInfo(
            "soupForContacts",
            "Contact",
            Arrays.asList("LastName"))
    )
);
```

Example:

```
{
  "syncs": [
    {
      "syncName": "myBriefcaseSyncDown",
      "syncType": "syncDown",
      "soupName": "does-not-matter",
      "target": {
        "type": "briefcase",
        "infos": [
          {
            "sobjectType": "Account",
            "fieldlist": [
              "Name",
              "Description"
            ],
            "soupName": "accounts"
          },
          {
            "sobjectType": "Contact",
            "fieldlist": [
              "LastName"
            ],
            "soupName": "contacts"
          }
        ]
      },
      "options": {
        "mergeMode": "OVERWRITE"
      }
    }
  ]
}
```

See Also

- Invoking the Sync Down Method with a Custom Target
- "Configure a Briefcase" in the *Briefcase Builder* online help

Syncing Metadata and Layouts

Mobile SDK 6.2 introduces new API features that simplify object discovery and presentation. These features harness the power of Mobile Sync to access Salesforce object metadata and layouts. Mobile SDK automatically stores the data in predefined soups for offline use and structured data models for easy querying.

When you use the Mobile SDK metadata and layout APIs, Mobile Sync creates and sends a REST request on your behalf and returns the data to your app. Instead of returning a raw JSON representation, however, the APIs format the response in custom data model objects. To enable offline use, these APIs store the response data locally in SmartStore soups. The feature itself creates the necessary soups and populates them through internal sync down targets.

To use this feature, you call a single method—one for metadata, one for layouts—and implement a callback handler for the response.

What Can You Do with Metadata?

Object metadata discloses the structure of a requested sObject or custom object type. Using metadata, your app can examine the object's field list, for example, to build valid queries at runtime. Or you can directly get field configuration properties, relationship graphs, URLs, and other object data from the response. Internally, metadata classes call the Salesforce "describe" API.

What Can You Do with Layouts?

Layouts provide JSON structures that define a standard configuration of labels and fields for objects and search results. You can use this configuration as a preformatted design spec for screens that display the object's data. Internally, the Mobile SDK layout API calls the Salesforce "record layout" API.

API Implementation

Mobile SDK defines several levels of iOS and Android native classes to sync metadata and layouts.

- *Sync manager classes* define the methods you call to obtain metadata and layouts.
  - **iOS:**

Swift	Objective-C
`MetadataSyncManager`	`SFMetadataSyncManager`
`LayoutSyncManager`	`SFLayoutSyncManager`

  - **Android:** `MetadataSyncManager`, `LayoutSyncManager`

  Sync manager objects create the following SmartStore soups and populate them during sync down.

  - `sfdcMetadata`
  - `sfdcLayouts`

  These soups are indexed for offline efficiency.

- *Data model classes* are structured containers for the metadata and layouts returned to your callback block.

- **iOS:**

Swift	Objective-C
`Metadata`	`SFMetadata`
`Layout`	`SFLayout`

- **Android:** `Metadata`, `Layout`

- *Custom sync down target classes* handle Salesforce queries and SmartStore soup synchronization behind the scenes.
  - **iOS:**

Swift	Objective-C
`MetadataSyncDownTarget`	`SFMetadataSyncDownTarget`
`LayoutSyncDownTarget`	`SFLayoutSyncDownTarget`

,

  - **Android:** `MetadataSyncDownTarget`, `LayoutSyncDownTarget`

Syncing Metadata and Layouts on iOS

Metadata and layout syncing on iOS is easy to use. To get started, learn how to initialize and configure the APIs.

Initializing Metadata and Layout Sync Managers

In iOS, metadata and layout managers are both shared objects. You access them by calling the `sharedInstance` class method as follows:

Swift

```
MetadataSyncManager.sharedInstance()
LayoutSyncManager.sharedInstance()
```

Objective-C

```
[SFMetadataSyncManager sharedInstance]
[SFLayoutSyncManager sharedInstance]
```

In this form, `sharedInstance` initializes the manager with the current user's credentials and the default store.

In multi-user environments, you can also initialize the manager with a logged-in but non-current user.

Swift

```
MetadataSyncManager.sharedInstance(user)
LayoutSyncManager.sharedInstance(user)
```

Objective-C

```
[SFMetadataSyncManager sharedInstance:user]
[SFLayoutSyncManager sharedInstance:user]
```

Here, `user` is an instance of `UserAccount` (Swift) or `SFUserAccount` (Objective-C).

To specify a store other than the user's default, use

Swift

```
MetadataSyncManager.sharedInstance(user, smartStore: store)
LayoutSyncManager.sharedInstance(user, smartStore: store)
```

Objective-C

```
[SFMetadataSyncManager sharedInstance:user smartStore:store]
[SFLayoutSyncManager sharedInstance:user smartStore:store]
```

To tell the manager to default to the current user, set `user` to nil. The `store` argument is an instance of `SmartStore` (Swift) or `SFSmartStore` (Objective-C) and must be associated with the given user. When a valid store is provided, Mobile Sync uses the given store to create its metadata and layout soups.

Retrieving Metadata (iOS)

You use metadata manager classes to fetch metadata from a Salesforce org or a SmartStore instance. To fetch, call the following asynchronous method on the shared instance of your metadata sync manager.

Swift

```
MetadataSyncManager.sharedInstance().fetchMetadata(forObject: String,
                                                   mode: FetchMode,
                                          completionBlock: MetadataSyncCompletionBlock)
```

Objective-C

```
- (void)fetchMetadataForObject:(nonnull NSString *)objectType
                          mode:(SFSDKFetchMode)mode
               completionBlock:(nonnull SFMetadataSyncCompletionBlock)completionBlock;
```

`objectType`
The Salesforce object whose metadata you're fetching. For example, "Account" or "Opportunity".

`mode`
This parameter helps determine the data's source location. Data retrieval modes include:

- – iOS (Swift): `FetchMode.cacheOnly`
  – iOS (Objective-C): `SFSDKFetchModeCacheOnly`
  – Android: `CACHE_ONLY`

  Fetches data from the cache. If cached data is not available, returns null.
- – iOS (Swift): `FetchMode.cacheFirst`
  – iOS (Objective-C): `SFSDKFetchModeCacheFirst`
  – Android: `CACHE_FIRST`

  Fetches data from the cache. If cached data is not available, fetches data from the server .
- – iOS (Swift): `FetchMode.serverFirst`
  – iOS (Objective-C): `SFSDKFetchModeServerFirst`
  – Android: `SERVER_FIRST`

  Fetches data from the server. If server data is not available, fetches data from the cache. Data fetched from the server is automatically cached.

`completionBlock`
Callback block that executes asynchronously when the operation completes. You pass the block's implementation or handle to this parameter. This block implements the following method prototype:

```
typedef void (^SFMetadataSyncCompletionBlock) (SFMetadata * _Nullable metadata);
```

Mobile SDK passes a metadata object to this callback method. This object contains the true data model of the requested Salesforce object. You can use this metadata to query specific fields. This class defines properties whose names match the field names in the object's manifest. Class properties represent all custom fields and customizable standard fields.

Retrieving Layouts (iOS)

To sync layouts, call the following asynchronous method on the shared instance of your layout sync manager.

Swift

Objective-C

```
- (void)fetchLayoutForObjectAPIName:(nonnull NSString *)objectAPIName
                         formFactor:(nullable NSString *)formFactor
                         layoutType:(nullable NSString *)layoutType
                               mode:(nullable NSString *)mode
                       recordTypeId:(nullable NSString *)recordTypeId
                           syncMode:(SFSDKFetchMode)syncMode
                  completionBlock:(nonnull SFLayoutSyncCompletionBlock)completionBlock;
```

`objectAPIName`
(Required) Salesforce object whose layout you're fetching. For example, "Account" or "Opportunity".

`formFactor`
(Optional) Form factor of the layout you're fetching. Supported values are "Large", "Medium", and "Small". If not specified, defaults to "Large".

`layoutType`
(Optional) Type of layout you're fetching. Supported values are "Compact" and "Full". If not specified, defaults to "Full".

`mode`
(Optional) Record mode of the layout you're fetching. Supported values are "Create", "Edit", and "View". If not specified, defaults to "View".

`recordTypeId`
(Optional) Record type whose layout you're fetching. If not specified, uses the default record type.

`syncMode`
Retrieval mode to use while retrieving data. Supported values are:

- `SFSDKFetchModeCacheOnly`—Fetches data from the cache. If cached data is not available, returns null.
- `SFSDKFetchModeCacheFirst`—Fetches data from the cache. If cached data is not available, fetches data from the server.
- `SFSDKFetchModeServerFirst`—Fetches data from the server. If server data is not available, fetches data from the cache. Data fetched from the server is automatically cached.

- **`completionBlock`**
  Asynchronous block that is triggered when the operation completes. You pass either the block's implementation or its handle to this parameter. This block implements the following method prototype:

  ```
  typedef void (^SFLayoutSyncCompletionBlock) (NSString * _Nonnull objectAPIName,
      NSString * _Nullable formFactor,
  ```

```
    NSString * _Nullable layoutType,
    NSString * _Nullable mode,
    NSString * _Nullable recordTypeId,
    SFLayout  * _Nullable layout);
```

Mobile SDK passes an `SFLayout` object to this callback method. This object contains the true data model of the requested Salesforce object's layout. You can use this object's properties to query specific fields.

Behind the scenes, `SFLayoutSyncManager` uses a Mobile Sync `SFLayoutSyncDownTarget` object to automatically create a SmartStore soup that contains the returned data. This soup includes index specs for retrieving layout data when the device is offline.

Syncing Metadata and Layouts on Android

Metadata and layout syncing on Android is easy to use. To get started, learn how to initialize and configure the APIs.

Initializing MetadataSyncManager and LayoutSyncManager

In Android, you access a metadata or layout manager by calling the `getInstance()` static method as follows.

```
MetadataSyncManager.getInstance();
LayoutSyncManager.getInstance();
```

In this form, `getInstance()` intializes the manager with the current user's credentials and default store.

In multi-user environments, you can also initialize the metadata manager with a logged-in but non-current user by calling

```
MetadataSyncManager.getInstance(account);
LayoutSyncManager.getInstance(account);
```

In this form, `account` is an instance of `UserAccount`. The `getInstance()` method intializes the manager with the given user's credentials and default store.

To specify a community that the given user belongs to, provide its ID by calling

```
MetadataSyncManager.getInstance(account, id);
LayoutSyncManager.getInstance(account, id);
```

If `account` is null, the manager defaults to the current user.

To specify a named SmartStore instance that's associated with the given user, use

```
MetadataSyncManager.getInstance(account, id, store);
LayoutSyncManager.getInstance(account, id, store);
```

To tell the manager to ignore the community setting, set `id` to null. The `store` argument is an instance of `SmartStore` and must be associated with the given user. When a valid store is provided, Mobile Sync uses the given store to create its metadata and layout soups.

Retrieving Metadata

You use metadata manager classes to fetch metadata from a Salesforce org or a SmartStore instance. To fetch, call the following asynchronous method on your `MetadataSyncManager` instance.

```
public void fetchMetadata(String objectType, Constants.Mode mode, MetadataSyncCallback
syncCallback);
```

objectType
The Salesforce object whose metadata you're fetching. For example, "Account" or "Opportunity".

mode
This parameter helps determine the data's source location. Data retrieval modes include:

- `CACHE_ONLY` (Android) or `SFSDKFetchModeCacheOnly` (iOS) - Fetches data from the cache. If cached data is not available, returns null.
- `CACHE_FIRST` (Android) or `SFSDKFetchModeCacheFirst` (iOS) - Fetches data from the cache. If cached data is not available, fetches data from the server .
- `SERVER_FIRST` (Android) or `SFSDKFetchModeServerFirst` (iOS) Fetches data from the server. If server data is not available, fetches data from the cache. Data fetched from the server is automatically cached.

syncCallback
Asynchronous block that is executed when the operation completes. You pass the block's implementation or handle to this parameter. This block takes the form of a `MetadataSyncCallback` interface that defines a single method:

```
void onSyncComplete(Metadata metadata);
```

Mobile SDK passes a `Metadata` object to this callback method. This object contains the true data model of the requested Salesforce object. You can use this metadata to query specific fields. This class defines properties whose names match the field names in the object's manifest. Class properties represent all custom fields and customizable standard fields.

Retrieving Layouts (Android)

To sync layouts, call the following asynchronous method on your `LayoutSyncManager` instance.

```
public void fetchLayout(String objectAPIName, String formFactor,
    String layoutType, String mode, String recordTypeId,
    Constants.Mode syncMode, LayoutSyncCallback syncCallback);
```

objectAPIName
(Required) Salesforce object whose layout you're fetching. For example, "Account" or "Opportunity".

formFactor
(Optional) Form factor of the layout you're fetching. Supported values are "Large", "Medium", and "Small". If not specified, defaults to "Large".

layoutType
(Optional) Type of layout you're fetching. Supported values are "Compact" and "Full". If not specified, defaults to "Full".

mode
(Optional) Record mode of the layout you're fetching. Supported values are "Create", "Edit", and "View". If not specified, defaults to "View".

recordTypeId
(Optional) Record type whose layout you're fetching. If not specified, uses the default record type.

syncMode
Retrieval mode to use while retrieving data. Supported values are:

- `CACHE_ONLY`—Fetches data from the cache. If cached data is not available, returns null.
- `CACHE_FIRST`—Fetches data from the cache. If cached data is not available, fetches data from the server.
- `SERVER_FIRST`—Fetches data from the server. If server data is not available, fetches data from the cache. Data fetched from the server is automatically cached.

- **`syncCallback`**
  Callback method that executes asynchronously when the operation completes. You pass either the method's implementation or its handle to this parameter. This block implements the following method prototype:

  ```
  void onSyncComplete(String objectAPIName, String formFactor, String layoutType,
      String mode, String recordTypeId, Layout layout);
  ```

Mobile SDK passes an `SFLayout` object to this callback method. This object contains the true data model of the requested Salesforce object's layout. You can use this object's properties to query specific fields.

Behind the scenes, `LayoutSyncManager` uses a Mobile Sync `LayoutSyncDownTarget` object to automatically create a SmartStore soup that contains the returned data. This soup includes index specs for retrieving layout data when the device is offline.

Using the Refresh Sync Down Target

Many apps download records, cache all of them, and then let users edit them from the SmartStore cache when connectivity drops. Local "offline" work is quick and efficient—a great user experience—but, when connectivity resumes, it's important to refresh the cached records with server updates.

To maximize performance and efficiency, Mobile SDK provides a *refresh* sync down target. The refresh target supports a single call that doesn't require preparatory coding. You create an instance of the target with a soup name, an object type, and a list of fields. You then pass the target instance to a sync down method. The refresh target gathers IDs of the pertinent soup records, queries the server for the current field values, and then refreshes the soup.

Refresh Target APIs

The refresh sync down target is available on iOS and Android for native, React native, and hybrid apps.

iOS

Class:

Swift	Objective-C
`RefreshSyncDownTarget`	`SFRefreshSyncDownTarget`

Factory method:

Swift

```
RefreshSyncDownTarget.newSyncTarget(soupName:objectType:fieldList:)
```

Here's an example:

```
let refreshTarget = RefreshSyncDownTarget.newSyncTarget("MySoup", objectType: "Contact",
 fieldList: ["Id","Name"])
```

Objective-C

```
+ (SFRefreshSyncDownTarget*) newSyncTarget:(NSString*)soupName
objectType:(NSString*)objectType fieldlist:(NSArray*)fieldlist
```

Android

Class:

```
com.salesforce.androidsdk.mobilesync.util.RefreshSyncDownTarget
```

Constructor:

```
public RefreshSyncDownTarget(List<String> fieldlist,
    String objectType, String soupName)
```

JavaScript (Hybrid, React Native)

Function:

```
var target = {soupName:xxx, type:"refresh",
    sobjectType:yyy, fieldlist:["Id", ...]};
```

Using the Batch Sync Up Target

To enhance performance in large sync up operations, Mobile SDK 7.1 introduces a batch sync up target.

iOS Native	Android Native

Swift
`BatchSyncUpTarget`
Objective-C
`SFBatchSyncUpTarget`
	`BatchSyncUpTarget`

This target enhances the standard sync up target behavior by calling the Salesforce composite API. The composite API sends local records to the server in batches of up to 25 records.

iOS Native

Swift

```
var target = BatchSyncUpTarget.init(createFieldList, updateFieldList)
let syncMgr = SyncManager.sharedInstance(store: self.store!)
syncMgr!.syncUp(target: target, options: options, soupName: soupName) {...}
```

Objective-C

```
SFBatchSyncUpTarget* target = [[SFBatchSyncUpTarget alloc] init];
// or
SFBatchSyncUpTarget* target =
    [[SFBatchSyncUpTarget alloc] initWithCreateFieldlist:createList
                                          updateFieldlist:updateList];
[syncManager syncUpWithTarget:target
                      options:options
                   soupName:soupName
                updateBlock:updateBlock];
```

Android Native

```
BatchSyncUpTarget target = new BatchSyncUpTarget();
// or
BatchSyncUpTarget target = new BatchSyncUpTarget(createFieldlist, updateFieldlist);
syncManager.syncUp(target, options, soupName, callback);
```

Hybrid and React Native

Existing sync up targets in hybrid and React Native apps automatically use the batch sync up target. To use a different sync up target implementation such as the legacy `SyncUpTarget` class, specify "androidImpl" or "iOSImpl".

Usage in Sync Config Files

By default, sync up targets defined in sync config files use batch APIs. For example, the following sync configuration creates a batch sync up target.

```
{ "syncs": [
    {
        "syncName": "syncUpContacts",
        "syncType": "syncUp",
        "soupName": "contacts",
        "target":
             {"createFieldlist":
                 ["FirstName",
                  "LastName",
                  "Title",
                  "MobilePhone",
                  "Email",
                  "Department",
                  "HomePhone"]
             },
        "options":
             {"fieldlist":
                 ["Id",
                  "FirstName",
                  "LastName",
                  "Title",
                  "MobilePhone",
                  "Email",
                  "Department",
                  "HomePhone"],
        "mergeMode":"LEAVE_IF_CHANGED"}
    }
]}
```

See Also

- Invoking the Sync Up Method with a Custom Target
- "Composite" in *REST API Developer Guide*

Using the sObject Collection Sync Up Target

For the very best performance in large sync up operations, Mobile SDK 10.1 introduced an sObject collection sync up target.

iOS Native	Android Native

Swift
`CollectionSyncUpTarget`
Objective-C
`SFCollectionSyncUpTarget`
	`CollectionSyncUpTarget`

This target enhances the standard sync up target behavior by using the Salesforce sObject Collections API. This API sends local records to the server in batches of up to 200 records. This target can be up to five times faster than the Batch sync up target, and up to 10 times faster than the standard single record sync up target.

Note: Actual performance can vary depending on specific records, sObjects, and network conditions.

iOS Native

Objective-C

```
SFCollectionSyncUpTarget* target = [[SFCollectionSyncUpTarget alloc] init];
// or
SFCollectionSyncUpTarget* target =
    [[SFCollectionSyncUpTarget alloc] initWithCreateFieldlist:createList
                                                updateFieldlist:updateList];

[syncManager syncUpWithTarget:target
                      options:options
                   soupName:soupName
                updateBlock:updateBlock];
```

Android Native

```
CollectionSyncUpTarget target = new CollectionSyncUpTarget();
// or
CollectionSyncUpTarget target = new CollectionSyncUpTarget(createFieldlist, updateFieldlist);

syncManager.syncUp(target, options, soupName, callback);
```

Hybrid and React Native

Existing sync up targets in hybrid and React Native apps automatically use the sObject collection sync up target.

Note: This behavior replaces the automatic use of the Batch sync up target. The new sObject collection target is backwards compatible, and should result in improved performance without requiring any changes to your code.

To use a different sync up target implementation such as the legacy `SyncUpTarget` class, specify that class in the “androidImpl” or “iOSImpl” setting.

Usage in Sync Config Files

By default, sync up targets defined in sync config files use sObject Collection APIs. For example, the following sync configuration creates an sObject Collection sync up target.

```
{
  "syncs": [
    {
      "syncName": "syncUpContacts",
      "syncType": "syncUp",
      "soupName": "contacts",
      "target":
          {"createFieldlist":
              ["FirstName",
               "LastName",
               "Title",
               "MobilePhone",
               "Email",
               "Department",
               "HomePhone"]
          },
      "options":
          {"fieldlist":
              ["Id",
               "FirstName",
               "LastName",
               "Title",
               "MobilePhone",
               "Email",
               "Department",
               "HomePhone"],
           "mergeMode":"LEAVE_IF_CHANGED"
          }
      }
   ]
}
```

See Also

- Invoking the Sync Up Method with a Custom Target
- "sObject Collections" in *REST API Developer Guide*

Using Custom Sync Down Targets

During sync down operations, a sync down target controls the set of records to be downloaded and the request endpoint. You can use pre-formatted MRU, SOQL-based, and SOSL-based targets, or you can create custom targets. Custom targets can access arbitrary REST endpoints both inside and outside of Salesforce.

Defining a Custom Sync Down Target

You define custom targets for sync down operations by subclassing your platform's abstract base class for sync down targets. To use custom targets in hybrid apps, implement a custom native target class for each platform you support. The base sync down target classes are:

- **iOS:**

Swift	Objective-C
`SyncDownTarget`	`SFSyncDownTarget`

- **Android:** `SyncDownTarget`

Note: These classes sync the requested records but not their related records. To include related records, use the sync target classes described in Syncing Related Records.

Required Methods

Every custom target class must implement the following required methods.

Start Fetch Method

Called by the sync manager to initiate the sync down operation. If `maxTimeStamp` is greater than 0, this operation becomes a "resync". It then returns only the records that have been created or updated since the specified time.

iOS:

Swift

```
func startFetch(syncManager: SyncManager,
                     maxTimeStamp: Int64,
                       errorBlock: SyncDownErrorBlock,
                         complete: SyncDownCompleteBlock)
```

Objective-C

```
- (void) startFetch:(SFMobileSyncSyncManager*)syncManager
       maxTimeStamp:(long long)maxTimeStamp
         errorBlock:(SFSyncDownTargetFetchErrorBlock)
                    errorBlock
      completeBlock:(SFSyncDownTargetFetchCompleteBlock)
                    completeBlock;
```

Android:

```
JSONArray startFetch(SyncManager syncManager, long maxTimeStamp);
```

Continue Fetching Method

Called by the sync manager repeatedly until the method returns null. This process retrieves all records that require syncing.

iOS:

Swift

```
func continueFetch(syncManager: SyncManager,
                           onFail: SyncDownErrorBlock,
                       onComplete: SyncDownCompletionBlock?)
```

Objective-C

```
- (void)
    continueFetch:(SFMobileSyncSyncManager*)syncManager
       errorBlock:(SFSyncDownTargetFetchErrorBlock)
```

```
                errorBlock
    completeBlock:(SFSyncDownTargetFetchCompleteBlock)
                completeBlock;
```

Android:

```
JSONArray continueFetch(SyncManager syncManager);
```

modificationDateFieldName Property (Optional)

Optionally, you can override the `modificationDateFieldName` property in your custom class. If you provide `modificationDateFieldName`, Mobile Sync uses the field with this name to compute the `maxTimestamp` value that `startFetch` uses to resync the records. Default field name is `lastModifiedDate`.

iOS (Swift and Objective-C):

`modificationDateFieldName` property

Android:

```
String getModificationDateFieldName();
```

idFieldName Property (Optional)

If you provide `"idFieldName"`, Mobile Sync uses the field with the given name to get the ID of the record. For example, if you specify `"idFieldName":"AcmeId"`, Mobile Sync obtains the record's ID from the `AcmeId` field instead of the default `Id` field.

iOS (Swift and Objective-C):

`idFieldName` property

Android:

```
String getIdFieldName();
```

Invoking the Sync Down Method with a Custom Target

iOS:

Pass an instance of your custom `SFSyncDownTarget` class to the `SFMobileSyncSyncManager` sync down method:

Swift

```
func syncDown(target: SyncDownTarget,
              soupName: String,
              onUpdate: ()) -> SyncState
```

Objective-C

```
- (SFSyncState*)
    syncDownWithTarget:(SFSyncDownTarget*)target
               soupName:(NSString*)soupName
           updateBlock:
               (SFSyncSyncManagerUpdateBlock)updateBlock;
```

Android:

Pass an instance of your custom `SyncDownTarget` class to the `SyncManager` sync down method:

```
SyncState syncDown(SyncDownTarget target, SyncOptions options, String soupName,
SyncUpdateCallback callback);
```

Hybrid:

1. Create a target object with the following property settings:

- Set `type` to "custom".
- Set at least one of the following properties:

  iOS (if supported):
  Set `iOSImpl` to the name of your iOS custom class.

  Android (if supported):
  Set `androidImpl` to the package-qualified name of your Android custom class.

The following example supports both iOS and Android:

```
var target =
{type:"custom",
 androidImpl:
 "com.salesforce.samples.notesync.ContentSoqlSyncDownTarget",
 iOSImpl:"SFContentSoqlSyncDownTarget",
 …
};
```

2. Pass this target to the hybrid sync down method:

```
cordova.require("com.salesforce.plugin.MobileSync").syncDown(target, …);
```

Sample Apps

iOS
The NoteSync native iOS sample app defines and uses the `SFContentSoqlSyncDownTarget` sync down target.

Android
The NoteSync native Android sample app defines and uses the `com.salesforce.samples.notesync.ContentSoqlSyncDownTarget` sync down target.

Using Custom Sync Up Targets

During sync up operations, a sync up target controls the set of records to be uploaded and the REST endpoint for updating records on the server. You can access arbitrary REST endpoints—both inside and outside of Salesforce—by creating custom sync up targets.

Defining a Custom Sync Up Target

You define custom targets for sync up operations by subclassing your platform's abstract base class for sync up targets. To use custom targets in hybrid apps, you're required to implement a custom native target class for each platform you support. The base sync up target classes are:

- **iOS:**

Swift	Objective-C
`SyncUpTarget`	`SFSyncUpTarget`

- **Android:** `SyncUpTarget`

Note: These classes sync the requested records but not their related records. To include related records, use the sync target classes described in Syncing Related Records.

Handling Field Lists in Create and Update Operations

A target's Create On Server and Update On Server methods operate only on the list of fields specified in their argument lists. However, a Salesforce object can require the target to create certain fields that can't be updated by apps. With these objects, a target that uses a single field list for both create and update operations can fail if it tries to update locked fields.

To specify distinct field lists for create and update operations, you can use an initializer method that supports `createFieldlist` and `updateFieldlist` parameters. This option can save you the effort of defining a custom target if you're doing so only to provide these field lists.

iOS

Call the following `SFSyncUpTarget` init method:

Swift

```
SyncUpTarget.init(createFieldlist: [Any]?, updateFieldlist: [Any]?)
```

Here's an example:

```
SyncUpTarget.init(createFieldlist: nil, updateFieldlist: ["Name"])
```

Objective-C

```
- (instancetype)initWithCreateFieldlist:(NSArray *)createFieldlist
                         updateFieldlist:(NSArray *)updateFieldlist
```

If you provide the `createFieldlist` and `updateFieldlist` arguments, the target uses them where applicable. In those cases, the target ignores the field list defined in the sync options object.

Android

Use the following `SyncUpTarget` constructor:

```
public SyncUpTarget(List<String> createFieldlist, List<String> updateFieldlist)
```

If you provide the `createFieldlist` and `updateFieldlist` arguments, the target uses them where applicable. In those cases, the target ignores the field list defined in the `SyncOptions` object.

Required Methods

Every custom target class must implement the following required methods.

Create On Server Method

Sync up a locally created record. Hybrid and React native apps can override the `fields` parameter by calling `syncUp` with the optional `createFieldList` parameter.

iOS:

Swift

```
func createOnServer(syncManager: SyncManager,
                        record: [AnyHashable : Any],
                     fieldlist: [Any],
                    onComplete: SyncUpcompletionBlock([AnyHashable : Any]?)
                                    -> Void,
                        onFail: SyncUpErrorBlock(Error) -> Void)
```

Objective-C

```
- (void) createOnServer:(NSString*)objectType
                 fields:(NSDictionary*)fields
```

```
        completionBlock:(SFSyncUpTargetCompleteBlock)
                        completionBlock
              failBlock:(SFSyncUpTargetErrorBlock)failBlock;
```

Android:

```
String createOnServer(SyncManager syncManager,
    String objectType, Map<String, Object> fields);
```

Update On Server Method

Sync up a locally updated record. For the `objectId` parameter, Mobile Sync uses the field specified in the `getIdFieldName()` method (Android) or the `idFieldName` property (iOS) of the custom target. Hybrid and React native apps can override the `fields` parameter by calling `syncUp` with the optional `updateFieldList` parameter.

iOS:

Swift

```
func updateOnServer(syncManager: SyncManager,
                          record: [AnyHashable : Any],
                       fieldlist: [Any],
                      onComplete: SyncUpcompletionBlock([AnyHashable : Any]?)
                                      -> Void,
                          onFail: SyncUpErrorBlock(Error) -> Void)
```

Objective-C

```
- (void) updateOnServer:(NSString*)objectType
               objectId:(NSString*)objectId
                 fields:(NSDictionary*)fields
        completionBlock:(SFSyncUpTargetCompleteBlock)
                        completionBlock
              failBlock:(SFSyncUpTargetErrorBlock)failBlock;
```

Android:

```
updateOnServer(SyncManager syncManager, String objectType, String objectId,
Map<String, Object> fields);
```

Delete On Server Method

Sync up a locally deleted record. For the `objectId` parameter, Mobile Sync uses the field specified in the `getIdFieldName()` method (Android) or the `idFieldName` property (iOS) of the custom target.

iOS:

Swift

```
func deleteOnServer(syncManager: SyncManager,
                          record: [AnyHashable : Any],
                       fieldlist: [Any],
                      onComplete: SyncUpcompletionBlock([AnyHashable : Any]?)
                                      -> Void,
                          onFail: SyncUpErrorBlock(Error) -> Void)
```

Objective-C

```
- (void) deleteOnServer:(NSString*)objectType
               objectId:(NSString*)objectId
        completionBlock:(SFSyncUpTargetCompleteBlock)
```

```
                          completionBlock
                failBlock:(SFSyncUpTargetErrorBlock)failBlock;
```

Android:

```
deleteOnServer(SyncManager syncManager, String objectType,
    String objectId);
```

Optional Configuration Changes

Optionally, you can override the following values in your custom class.

getIdsOfRecordsToSyncUp

List of record IDs returned for syncing up. By default, these methods return any record where `__local__` is true.

iOS:

Swift

```
func getIdsOfRecords(toSyncUp: SyncManager, soupName: String)
```

Objective-C

```
- (NSArray*)
getIdsOfRecordsToSyncUp:(SFMobileSyncSyncManager*)syncManager
               soupName:(NSString*)soupName;
```

Android:

```
Set<String> getIdsOfRecordsToSyncUp(SyncManager syncManager,
    String soupName);
```

Modification Date Field Name

Field used during a `LEAVE_IF_CHANGED` sync up operation to determine whether a record was remotely modified. Default value is `lastModifiedDate`.

iOS (Swift and Objective-C):

`modificationDateFieldName` property

Android:

```
String getModificationDateFieldName();
```

isNewerThanServer

Determines whether a soup element is more current than the corresponding server record.

iOS:

Swift

```
func isNewerThanServer(syncManager: SyncManager,
                              record: [AnyHashable : Any],
                       resultBlock: RecordNewerThanServerBlock(Bool) -> Void)
```

Objective-C

```
- (void)isNewerThanServer:(SFMobileSyncSyncManager *)syncManager
                   record:(NSDictionary*)record
              resultBlock:(SFSyncUpRecordNewerThanServerBlock)resultBlock;
```

Android:

```
public boolean isNewerThanServer(SyncManager syncManager,
    JSONObject record) throws JSONException, IOException
```

ID Field Name

Field used to get the ID of the record. For example, during sync up, Mobile SDK obtains the ID that it passes to the `updateOnServer()` method from the field whose name matches `idFieldName` in the local record.

iOS (Swift and Objective-C):

`idFieldName` property

Android:

```
String getIdFieldName();
```

Invoking the Sync Up Method with a Custom Target

iOS:

On a sync manager instance, call:

Swift

```
func syncUp(target: SyncUpTarget,
             options: SyncOptions,
            soupName: String,
            onUpdate: ()) -> SyncState
```

Here's an example:

```
let syncState = syncManager.syncUp(target: target, options: options,
    soupName: CONTACTS_SOUP, onUpdate: updateFunc)
```

Objective-C

```
- (SFSyncState*)
    syncUpWithTarget:(SFSyncUpTarget*)target
         syncOptions:(SFSyncOptions*)options
            soupName:(NSString*)soupName
         updateBlock:
              (SFSyncSyncManagerUpdateBlock)updateBlock;
```

Android:

On a `SyncManager` instance, call:

```
SyncState syncUp(SyncUpTarget target,
    SyncOptions options, String soupName,
    SyncUpdateCallback callback);
```

Hybrid:

```
cordova.require("com.salesforce.plugin.mobilesync").
      syncUp(isGlobalStore, target, soupName,
        options, successCB, errorCB);
cordova.require("com.salesforce.plugin.mobilesync").
      syncUp(storeConfig, target, soupName,
        options, successCB, errorCB);
```

Syncing Related Records

It's a common problem in syncing offline data: You can easily sync your explicit changes, but how do you update affected related records? You can do it manually with enough knowledge, determination, and perspicacity, but that's the old way. Starting with Mobile SDK 5.2, Mobile Sync provides tools that let you sync parent records and their related records with a single call.

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

Supported Relationship Types

Related record sync supports two types of one-to-many relationships: Lookup and Master-detail. Both types are defined in the child-to-parent, many-to-one direction. Each child knows its parent, but the parent doesn't know its children.

Lookup Relationships

A lookup relationship is a "loose" link between objects based on one or more fields. For lookup relationships:

- Child records don't require a parent field.
- Changes to one object don't affect the security, access, or deletion of a linked object.

Salesforce supports up to 25 lookup fields per object. Lookup relationships can be multiple layers deep.

Master-detail Relationships

A master-detail relationship is a parent-child link in which the parent object exerts some control over its children. In master-detail linkage:

- Child records require a parent field.
- The parent's access level determines the access level of its children.
- If a parent record is deleted, its children are also deleted.

Salesforce supports up to two master-detail fields per object, and up to three levels of master-detail relationships.

See Object Relationships Overview in Salesforce Help.

Note: Mobile Sync doesn't support many-to-many relationships.

Objects Used in Related Record Sync

Related record sync uses two special types of sync targets:

Parent-children sync up target

- **iOS:** `SFParentChildrenSyncUpTarget`
- **Android:** `ParentChildrenSyncUpTarget`

Handles locally created or updated records and deleted records.

Parent-children sync down target

- **iOS:** `SFParentChildrenSyncDownTarget`
- **Android:** `ParentChildrenSyncDownTarget`

Supports `resync` and `cleanResyncGhosts` methods.

These targets support *leave-if-changed* and *overwrite* merge modes. Each target provides a factory method (on iOS) or a constructor (on Android) that initializes an instance with the required information you provide. To perform the sync, you configure a new target class instance and pass it to the standard Mobile Sync sync method that accepts a target object.

To initialize the targets, you also provide two helper objects that deliver necessary related record information:

Parent information object

- **iOS:**

  Swift

  ```
  ParentInfo
  ```

  Objective-C

  ```
  SFParentInfo
  ```

- **Android:** `ParentInfo`

Includes:

- Object type
- Soup name
- ID field name (Optional in all cases. Defaults to "Id". Set this value only if you're specifying a different field to identify the records.)
- Last Modified Date field name (Optional in all cases. Defaults to "LastModifiedDate". Set this value only if you're specifying a different field for timestamps.)
- External ID field name (Optional in all cases. If provided, and if a locally created parent or child record specifies a value for it, Mobile Sync performs an `upsert` instead of `create`).

Child information object

- **iOS:** `ChildrenInfo`
- **Android:** `ChildrenInfo`

Includes:

- Object type
- Soup name
- Parent ID field name
- ID field name (Optional in all cases. Defaults to "Id". Set this value only if you're specifying a different field to identify the records.)
- Last Modified Date field name (Optional in all cases. Defaults to "LastModifiedDate". Set this value only if you're specifying a different field for timestamps.)
- External ID field name (Optional in all cases. If provided, and if a locally created parent or child record specifies a value for it, Mobile Sync performs an `upsert` instead of `create`).

Preparing Your SmartStore Data Model

To prepare for handling related objects offline, you first set up a SmartStore soup for each expected parent and child object type. For each soup, add indexed ID fields that model the server-side relationships. Here's the list of required indexed fields:

Soup for a parent object:
Field for server ID of record

Soup for a child object:
Field for server ID of record

Field for server ID of parent record

All sync operations—up and down—begin with the parent soup and then continue to the child soups. Here's how this flow works:

- When you sync down related records, you are targeting parent records, and the sync downloads those records and all their children. For example, if you're syncing accounts with their contacts, you get the contacts linked to the accounts you've downloaded. You don't get contacts that aren't linked to those accounts.

- When you sync up related records, Mobile Sync iterates over the soup of parent records and picks up related children records. Modified child records that aren't related to a parent record are ignored during that sync up operation.

Sync Up

To initialize the parent-child sync-up target, you provide parent information objects and child information objects. These objects fully describe the relationships between parent-child records, both on the server and in the local store. You also provide the list of fields to sync. Here's the full list of required information:

- Parent information object
- Child information object
- Relationship type (for example, master-detail or lookup)
- Fields to sync up in parent and children soups

The sync up operation iterates over the soup containing the parent records and uses the given information to pull related records from the children's soups. A record is considered dirty when the `__local__` field is set to true. A record tree—consisting of one parent and its children—is a candidate for sync up when any record in the tree is dirty. Whether the sync up actually occurs depends on how Mobile Sync handles the merge mode.

"Leave-if-changed" Merge Mode Handling

Mobile Sync fetches Last Modified Date fields of the target parent and children server records. The sync up operation skips that record tree if

- the last modified date of any fetched server record is more recent than its corresponding local record

  or

- if any fetched server record has been deleted.

Updates Applied after Sync Up

After the local changes have been synced to the server, Mobile Sync cleans up related records.

- If sync up creates any parent or child record on the server, Mobile Sync updates the server ID field of the corresponding local record. If the created record is a parent, the parent ID fields of its children are also updated in the local store.
- If any server records were deleted since the last sync down, Mobile Sync deletes the corresponding local records.
- If sync up deletes a parent record and the relationship type is master-detail, Mobile Sync deletes the record's children on the server and in the local store.

Sync Down

To initialize the parent-children sync-down target, you provide parent information objects and children information objects. These objects fully describe the relationships between parent-child records, both on the server and in the local store. You also provide a list of fields to sync and some SOQL filters.

The new sync down target is actually a subclass of the SOQL sync down target. Instead of being given the SOQL query, however, the target generates it from the parent and children information objects, list of fields, and SOQL filters.

Information Passed to Sync Down Targets

- Parent information object
- Child information object
- Relationship type (for example, master-detail or lookup)
- Fields to sync down in parent and children soups
- SOQL filter to apply in query on root records during sync down—for example, the condition of a WHERE clause

Server Call

Mobile Sync fetches the record trees, each consisting of one parent and its children, using SOQL. It then separates parents from their children and stores them in their respective soups.

"Leave-if-changed" Merge Mode Handling

Local record trees that contain any dirty records—locally created, modified, or deleted records—are left unaltered. For example, if a parent record has one dirty child, Mobile Sync doesn't update the parent or the child. This rule applies even if the parent is clean locally but has been changed on the server.

Handling Resync

During resync, Mobile Sync adjusts the SOQL query to download only those record trees in which the parent changed since the last sync.

Implementing Related Record Sync

After you understand the principles and requirements involved, implementing it is straightforward. You can add related record sync to your code in a few steps. The following code snippets demonstrate the technique using Account (parent) and Contact (child) objects.

Note: The following server-side limitations affect how you sync records from related and unrelated objects.

- You can update your soups using the Composite API. Be aware, however, that this API limits you to 25 records at a time. For an example, see the iOS `SFParentChildrenSyncUpTarget` or Android`ParentChildrenSyncUpTarget` implementation.
- SOQL limitations affect how Mobile Sync can select child records in a single query. As a result, `reSync` can sync the changed children of changed parents, but not the changed children of unchanged parents. To work around this limitation, you can use a separate custom target that directly queries child objects.
- SOQL doesn't have a UNION operator, so you can't get unrelated entities with a single call to the server. Instead, use separate queries. For example, you can use a distinct `SOQLSyncDownTarget` object for each query.
- To sync up non-related records, consider implementing a custom sync up target.

iOS and Android

1. Create a `userstore.json` file that defines SmartStore soups with the required indexed fields.

```
{
  "soups": [
    {
      "soupName": "ContactSoup",
      "indexes": [
        { "path": "Id", "type": "string"},
        { "path": "LastName", "type": "string"},
        { "path": "AccountId", "type": "string"},
        { "path": "__local__", "type": "string"},
        { "path": "__locally_created__", "type": "string"},
        { "path": "__locally_updated__", "type": "string"},
        { "path": "__locally_deleted__", "type": "string"}
      ]
    },
    {
      "soupName": "AccountSoup",
      "indexes": [
        { "path": "Id", "type": "string"},
        { "path": "Name", "type": "string"},
```

```
        { "path": "Description", "type": "string"},
        { "path": "__local__", "type": "string"},
        { "path": "__locally_created__", "type": "string"},
        { "path": "__locally_updated__", "type": "string"},
        { "path": "__locally_deleted__", "type": "string"}
      ]
    }
  ]
}
```

2. Create a `usersyncs.json` file that defines two named syncs. In this example, "DownSync" and "UpSync" configurations model the parent-child relationship between Account and Contact objects.

```
{
    "syncs": [{
        "syncName": "DownSync",
        "syncType": "syncDown",
        "soupName": "AccountSoup",
        "target": {
            "iOSImpl": "SFParentChildrenSyncDownTarget",
            "AndroidImpl": "ParentChildrenSyncDownTarget",
            "parent": {
                "idFieldName": "Id",
                "sobjectType": "Account",
                "modificationDateFieldName": "LastModifiedDate",
                "soupName": "AccountSoup"
            },
            "parentFieldlist": [
                "Id",
                "Name",
                "Description"
            ],
            "children": {
                "parentIdFieldName": "AccountId",
                "idFieldName": "Id",
                "sobjectType": "Contact",
                "modificationDateFieldName": "LastModifiedDate",
                "soupName": "ContactSoup",
                "sobjectTypePlural": "Contacts"
            },
            "childrenFieldlist": [
                "LastName",
                "AccountId"
            ],
            "relationshipType": "MASTER_DETAIL",
            "parentSoqlFilter": "Name LIKE 'A%' ",
            "type": "parent_children",
            "idFieldName": "Id"
        },
        "options": {"mergeMode": "OVERWRITE"}
    },

    {
        "syncName": "UpSync",
```

```
        "syncType": "syncUp",
        "soupName": "AccountSoup",
        "target": {
            "iOSImpl": "SFParentChildrenSyncUpTarget",
            "AndroidImpl": "ParentChildrenSyncUpTarget",
            "childrenCreateFieldlist": [
                "LastName",
                "AccountId"
            ],
            "parentCreateFieldlist": [
                "Id",
                "Name",
                "Description"
            ],
            "childrenUpdateFieldlist": [
                "LastName",
                "AccountId"
            ],
            "parentUpdateFieldlist": [
                "Name",
                "Description"
            ],
            "parent": {
                "idFieldName": "Id",
                "sobjectType": "Account",
                "modificationDateFieldName": "LastModifiedDate",
                "soupName": "AccountSoup"
            },
            "relationshipType": "MASTER_DETAIL",
            "type": "rest",
            "modificationDateFieldName": "LastModifiedDate",
            "children": {
                "parentIdFieldName": "AccountId",
                "idFieldName": "Id",
                "sobjectType": "Contact",
                "modificationDateFieldName": "LastModifiedDate",
                "soupName": "ContactSoup",
                "sobjectTypePlural": "Contacts"
            },
            "parentUpdateFieldlist": [
                "Name",
                "Description"
            ],
            "idFieldName": "Id"
        },
        "options": {"mergeMode":"LEAVE_IF_CHANGED"}
    }
    ]
}
```

The following steps demonstrate how to use these configurations to synchronize related Account (parent) and Contact (child) records.

iOS

1. In your Xcode project under **Build Phases** > **Copy Bundle Resources**, add the `userstore.json` and `usersyncs.json` files to your Xcode target.
2. Load the store and sync configurations. Call these loaders after Mobile SDK is initialized and before you call any SmartStore or Mobile Sync methods. For example, in your `AppDelegate` class, call these methods in the block you pass to `loginIfRequired`. Don't call either of these methods more than once.

   Swift

   ```
   func application(_ application: UIApplication, didFinishLaunchingWithOptions
   launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
       self.window = UIWindow(frame: UIScreen.main.bounds)
       self.initializeAppViewState()
       AuthHelper.loginIfRequired {
           MobileSyncSDKManager.shared.setupUserStoreFromDefaultConfig()
           MobileSyncSDKManager.shared.setupUserSyncsFromDefaultConfig()
           self.setupRootViewController()
       }
   ...
       return true
   }
   ```

   Objective-C

   ```
   - (BOOL)application:(UIApplication *)application
   didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
   {
       self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
       [self initializeAppViewState];
       [SFSDKAuthHelper loginIfRequired:^{
           [self setupRootViewController];
           [[MobileSyncSDKManager sharedManager] setupUserStoreFromDefaultConfig];
           [[MobileSyncSDKManager sharedManager] setupUserSyncsFromDefaultConfig];
       }];
   ...
       return YES;
   }
   ```

3. Call a Mobile Sync `reSync` method that takes a sync name. By passing sync names from your configuration file, you can use the `reSync(named:update:)` method for every sync up and sync down operation.

   Swift

   ```
   self.syncMgr.reSync(named: kSyncName) { [weak self] (syncState) in
       // Handle updates
   }
   ```

   Objective-C

   ```
   [self.syncMgr reSyncByName:kSyncUpName updateBlock:^(SFSyncState* sync) {
       // Handle updates
   }
   ```

Android

1. Place the `userstore.json` and `usersyncs.json` files in your project's `res/raw` folder.

2. Call the Mobile Sync sync method that takes a sync name. By passing sync names from your configuration file, you can use the `reSync(named:update:)` method for every sync up and sync down operation.

   For sync up and sync down:

```
//Resync a predefined sync by its name

syncManager.reSync(syncName,
    new SyncUpdateCallback() {
        @Override
        public void onUpdate(SyncState sync) {
            // Handle updates
        }
    });
```

Using Mobile Sync in Hybrid and React Native Apps

Mobile Sync for JavaScript is a Mobile SDK library that represents Salesforce objects as JavaScript objects. To use Mobile Sync in JavaScript, you create models of Salesforce objects and manipulate the underlying records just by changing the model data. If you perform a SOQL or SOSL query, you receive the resulting records in a model collection rather than as a JSON string.

In hybrid apps, Mobile SDK provides two options for using Mobile Sync.

- `com.salesforce.plugin.mobilesync`: The Mobile Sync plug-in offers basic "sync up" and "sync down" functionality. This plug-in exposes part of the native Mobile Sync library. For simple syncing tasks, you can use the plug-in to sync records rapidly in a native thread, rather than in the web view.
- `mobilesync.js`: The Mobile Sync JavaScript library provides a Force.SObject data framework for more complex syncing operations. This library is based on `backbone.js`, an open-source JavaScript framework that defines an extensible data modeling mechanism. To understand this technology, browse the examples and documentation at backbonejs.org.

A set of sample hybrid applications demonstrate how to use Mobile Sync. Sample apps in the `hybrid/SampleApps/AccountEditor/assets/www` folder demonstrate how to use the Force.SObject library in `mobilesync.js`:

- Account Editor (`AccountEditor.html`)
- User Search (`UserSearch.html`)
- User and Group Search (`UserAndGroupSearch.html`)

The sample app in the `hybrid/SampleApps/SimpleSync` folder demonstrates how to use the Mobile Sync plug-in.

Which Hybrid Version of Mobile Sync Should I Use?

The file `mobilesync.js`—the JavaScript version of Mobile Sync—and native Mobile Sync—available to hybrid apps through a Cordova plug-in—share a name, but they offer different advantages.

`mobilesync.js` is built on backbone.js and gives you easy-to-use model objects to represent single records or collections of records. It also provides convenient fetch, save, and delete methods. However, it doesn't give you true sync down and sync up functionality. Fetching records with an SObjectCollection is similar to the plug-in's `syncDown` method, but it deposits all the retrieved objects in memory. For that reason, it's not the best choice for moving large data sets. Furthermore, you're required to implement the sync up functionality yourself. The AccountEditor sample app demonstrates a typical JavaScript `syncUp()` implementation.

Native Mobile Sync doesn't return model objects, but it provides robust `syncUp` and `syncDown` methods for moving large data sets to and from the server.

You can also use the two libraries together. For example, you can set up a `Force.StoreCache` with `mobilesync.js`, sync data into it using the Mobile Sync plug-in, and then call fetch or save using `mobilesync.js`. You can then sync up from the same cache using the Mobile Sync plug-in, and it all works.

Both libraries provide the means to define your own custom endpoints, so which do you choose? The following guidelines can help you decide:

- Use custom endpoints from `mobilesync.js` if you want to talk to the server directly for saving or fetching data with JavaScript.
- If you talk only to SmartStore and get data into SmartStore using the Mobile Sync plug-in and then you don't need the custom endpoints in `mobilesync.js`. However, you must define native custom targets.

Note: `mobilesync.js` uses promises internally.

Mobile SDK promised-based APIs include:

- `force+promise.js`
- The `smartstoreclient` Cordova plugin (`com.salesforce.plugin.smartstore.client`)
- `mobilesync.js`

About Backbone Technology

The Mobile Sync library, `mobilesync.js`, provides extensions to the open-source Backbone JavaScript library. The Backbone library defines key building blocks for structuring your web application:

- Models with key-value binding and custom events, for modeling your information
- Collections with a rich API of enumerable functions, for containing your data sets
- Views with declarative event handling, for displaying information in your models
- A router for controlling navigation between views

Salesforce Mobile Sync extends the `Model` and `Collection` core Backbone objects to connect them to the Salesforce REST API. Mobile Sync also provides optional offline support through SmartStore, the secure storage component of the Mobile SDK.

To learn more about Backbone, see http://backbonejs.org/ and http://backbonetutorials.com/. You can also search online for "backbone javascript" to find a wealth of tutorials and videos.

Models and Model Collections

Two types of objects make up the Mobile Sync data framework:

- Models
- Model collections

Definitions for these objects extend classes defined in `backbone.js`, a popular third-party JavaScript framework. For background information, see http://backbonetutorials.com.

Models

Models on the client represent server records. In Mobile Sync, model objects are instances of `Force.SObject`, a subclass of the `Backbone.Model` class. `SObject` extends `Model` to work with Salesforce APIs and, optionally, with SmartStore.

You can perform the following CRUD operations on `SObject` model objects:

- Create
- Destroy

- Fetch
- Save
- Get/set attributes

In addition, model objects are observable: Views and controllers can receive notifications when the objects change.

Properties

`Force.SObject` adds the following properties to `Backbone.Model`:

`sobjectType`
Required. The name of the Salesforce object that this model represents. This value can refer to either a standard object or a custom object.

`fieldlist`
Required. Names of fields to fetch, save, or destroy.

`cacheMode`
Offline behavior.

`mergeMode`
Conflict handling behavior.

`cache`
For updatable offline storage of records. The Mobile Sync comes bundled with Force.StoreCache, a cache implementation that is backed by SmartStore.

`cacheForOriginals`
Contains original copies of records fetched from server to support conflict detection.

Examples

You can assign values for model properties in several ways:

- As properties on a `Force.SObject` instance.
- As methods on a `Force.SObject` sub-class. These methods take a parameter that specifies the desired CRUD action (“create”, “read”, “update”, or “delete”).
- In the options parameter of the `fetch()`, `save()`, or `destroy()` function call.

For example, these code snippets are equivalent.

```
// As properties on a Force.SObject instance
acc = new Force.SObject({Id:"<some_id>"});
acc.sobjectType = "account";
acc.fieldlist = ["Id", "Name"];
acc.fetch();
```

```
// As methods on a Force.SObject sub-class
Account = Force.SObject.extend({
  sobjectType: "account",
  fieldlist: function(method) { return ["Id", "Name"];}
});
```

```
Acc = new Account({Id:"<some_id>"});
acc.fetch();
```

```
// In the options parameter of fetch()
acc = new Force.SObject({Id:"<some_id>"});
acc.sobjectType = "account";
acc.fetch({fieldlist:["Id", "Name"]);
```

Model Collections

Model collections in Mobile Sync are containers for query results. Query results stored in a model collection can come from the server via SOQL, SOSL, or MRU queries. Optionally, they can also come from the cache via SmartSQL (if the cache is SmartStore), or another query mechanism if you use an alternate cache.

Model collection objects are instances of `Force.SObjectCollection`, a subclass of the `Backbone.Collection` class. `SObjectCollection` extends `Collection` to work with Salesforce APIs and, optionally, with SmartStore.

Properties

`Force.SObjectCollection` adds the following properties to `Backbone.Collection`:

`config`
Required. Defines the records the collection can hold (using SOQL, SOSL, MRU or SmartSQL).

`cache`
For updatable offline storage of records. The Mobile Sync comes bundled with Force.StoreCache, a cache implementation that's backed by SmartStore.

`cacheForOriginals`
Contains original copies of records fetched from server to support conflict detection.

Examples

You can assign values for model collection properties in several ways:

- As properties on a `Force.SObject` instance
- As methods on a `Force.SObject` sub-class
- In the options parameter of the `fetch()`, `save()`, or `destroy()` function call

For example, these code snippets are equivalent.

```
// As properties on a Force.SObject instance
list = new Force.SObjectCollection({config:<valid_config>});
list.fetch();
```

```
// As methods on a Force.SObject sub-class
MyCollection = Force.SObjectCollection.extend({
  config: function() { return <valid_config>; }
});
list = new MyCollection();
list.fetch();
```

```
// In the options parameter of fetch()
list = new Force.SObjectCollection();
list.fetch({config:valid_config});
```

Using the Mobile Sync Plugin

Beginning with Mobile SDK 3.0, the Mobile Sync plug-in provides JavaScript access to the native Mobile Sync library's "sync up" and "sync down" functionality. As a result, performance-intensive operations—network negotiations, parsing, SmartStore management—run on native threads that do not affect web view operations.

Adding the Mobile Sync plug-in to your hybrid project is a function of the Mobile SDK npm scripts:

- For forceios version 3.0 or later, the plug-in is automatically included.
- For forcedroid version 3.0 or later, answer "yes" when asked if you want to use SmartStore.

If you're adding the Mobile Sync plug-in to an existing hybrid app, it's best to re-create the app using the latest version of forcedroid or forceios. When the new app is ready, copy your custom HTML, CSS, and JavaScript files from your old project into the new project.

Mobile Sync Plugin Methods

The Mobile Sync plug-in exposes two methods: `syncDown()` and `syncUp()`. When you use these methods, several important guidelines can make your life simpler:

- To create, update, or delete records locally for syncing with the plug-in, use `Force.SObject` from `mobilesync.js`. Mobile Sync expects some special fields on soup records that `mobilesync.js` creates for you.
- Similarly, to create the soup that you'll use in your sync operations, use `Force.StoreCache` from `mobilesync.js`.
- If you've changed objects in the soup, always call `syncUp()` before calling `syncDown()`.

syncDown() Method

Downloads the sObjects specified by `target` into the SmartStore soup specified by `soupName`. If sObjects in the soup have the same ID as objects specified in the target, Mobile Sync overwrites the duplicate objects in the soup.

Mobile Sync also supports a refresh sync down target, which simplifies the process of refreshing cached records. See Using the Refresh Sync Down Target.

Syntax

```
cordova.require("com.salesforce.plugin.mobilesync").syncDown(
    [isGlobalStore, ]target, soupName, options, callback);
cordova.require("com.salesforce.plugin.mobilesync").syncDown(
    [storeConfig, ]target, soupName, options, callback);
```

Parameters

`isGlobalStore`
(Optional) Boolean that indicates whether this operation occurs in a global or user-based SmartStore database. Defaults to `false`.

`storeConfig`
(Optional) `StoreConfig` object that specifies a store name and whether the store is global or user-based.

`target`
Indicates which sObjects to download to the soup. Can be any of the following strings:

- `{type:"soql", query:"<soql query>"}`

  Downloads the sObjects returned by the given SOQL query.

- `{type:"sosl", query:"<sosl query>"}`

  Downloads the sObjects returned by the given SOSL query.

- `{type:"mru", sobjectType:"<sobject type>", fieldlist:"<fields to fetch>"}`

  Downloads the specified fields of the most recently used sObjects of the specified sObject type.

- `{type:"custom", androidImpl:"<name of native Android target class (if supported)>", iOSImpl:"<name of native iOS target class (if supported)>"}`

  Downloads the records specified by the given custom targets. If you use custom targets, provide either `androidImpl` or `iOSImpl`, or, preferably, both. See Using Custom Sync Down Targets.

`soupName`
Name of soup that receives the downloaded sObjects.

`options`
Use one of the following values:

- To overwrite local records that have been modified, pass `{mergeMode:Force.MERGE_MODE_DOWNLOAD.OVERWRITE}`.
- To preserve local records that have been modified, pass `{mergeMode:Force.MERGE_MODE_DOWNLOAD.LEAVE_IF_CHANGED}`. With this value, locally modified records are not overwritten.

`callback`
Function called once the sync has started. This function is called multiple times during a sync operation:

1. When the sync operation begins
2. When the internal REST request has completed
3. After each page of results is downloaded, until 100% of results have been received

Status updates on the sync operation arrive via browser events. To listen for these updates, use the following code:

```
document.addEventListener("sync",
    function(event) {
        // event.detail contains the status of the sync operation
    }
);
```

The `event.detail` member contains a map with the following fields:

- `syncId`: ID for this sync operation
- `type`: "syncDown"
- `target`: Targets you provided
- `soupName`: Soup name you provided
- `options`: "{}"
- `status`: Sync status, which can be "NEW", "RUNNING", "DONE" or "FAILED"
- `progress`: Percent of total records downloaded so far (integer, 0–100)
- `totalSize`: Number of records downloaded so far

syncUp() Method

Uploads created, deleted, or updated records in the SmartStore soup specified by `soupName`, and then updates, creates, or deletes the corresponding records on the Salesforce server. Updates are reported through browser events.

Syntax

```
cordova.require("com.salesforce.plugin.mobilesync").syncUp(isGlobalStore, target, soupName,
 options, callback);
cordova.require("com.salesforce.plugin.mobilesync").syncUp(storeConfig, target, soupName,
 options, callback);
```

Parameters

`isGlobalStore`
(Optional) Boolean that indicates whether this operation occurs in a global or user-based SmartStore database. Defaults to `false`.

`storeConfig`
(Optional) `StoreConfig` object that specifies a store name and whether the store is global or user-based.

`target`

JSON object that contains at least the name of one native custom target class, if you define custom targets.

A Salesforce object can require certain fields that can't be updated by apps. With these objects, a target that uses a single field list for both create and update operations can fail if it tries to update locked fields. Past versions of Mobile Sync required the developer to create a custom native target to differentiate between create and update field lists.

As of Mobile SDK 5.1, you no longer have to define custom native targets for these scenarios. Instead, to specify distinct field lists for create and update operations, add the following JSON object to the `target` object:

```
{createFieldlist: [<array_of_fields_to_create>], updateFieldlist:
[<another_array_of_fields_to_update>]}
```

If you provide `createFieldlist` and `updateFieldlist` arguments, the native custom target uses them where applicable. In those cases, the target ignores the field list defined in its "sync options" settings.

See the `syncDown()` method description for more information on `target` metadata.

`soupName`
Name of soup from which to upload sObjects.

`options`
A map with the following keys:

- `fieldlist`: List of fields sent to the server.
- `mergeMode`:
  - To overwrite remote records that have been modified, pass "OVERWRITE".
  - To preserve remote records that have been modified, pass "LEAVE_IF_CHANGED". With this value, modified records on the server are not overwritten.
  - Defaults to "OVERWRITE" if not specified.

`callback`
Function called multiple times after the sync has started. During the sync operation, this function is called for these events:

1. When the sync operation begins
2. When the internal REST request has completed
3. After each page of results is uploaded, until 100% of results have been received

Status updates on the sync operation arrive via browser events. To listen for these updates, use the following code:

```
document.addEventListener("sync",
    function(event) {
        // event.detail contains the status of the sync operation
    }
);
```

The `event.detail` member contains a map with the following fields:

- `syncId`: ID for this sync operation
- `type`: "syncUp"
- `target`: "{}" or a map or dictionary containing the class names of iOS and Android custom target classes you've implemented
- `soupName`: Soup name you provided
- `options`:
  - `fieldlist`: List of fields sent to the server
  - `mergeMode`: "OVERWRITE" or "LEAVE_IF_CHANGED"
- `status`: Sync status, which can be "NEW", "RUNNING", "DONE" or "FAILED"
- `progress`: Percent of total records downloaded so far (integer, 0–100)
- `totalSize`: Number of records downloaded so far

SEE ALSO:

Creating and Accessing User-based Stores

Using Mobile Sync in JavaScript

To use Mobile Sync in a hybrid app, import these files with <script> tags:

- `jquery-x.x.x.min.js` (use the version in the `dependencies/jquery/` directory of the SalesforceMobileSDK-Shared repository)
- `underscore-x.x.x.min.js` (use the version in the `dependencies/underscore/` directory of the SalesforceMobileSDK-Shared repository)
- `backbone-x.x.x.min.js` (use the version in the `dependencies/backbone/` directory of the SalesforceMobileSDK-Shared repository)
- `cordova.js`
- `force.js`
- `mobilesync.js`

Implementing a Model Object

To begin using Mobile Sync objects, define a model object to represent each `SObject` that you want to manipulate. The `SObjects` can be standard Salesforce objects or custom objects. For example, this code creates a model of the Account object that sets the two required properties—`sobjectType` and `fieldlist`—and defines a `cacheMode()` function.

```
app.models.Account = Force.SObject.extend({
    sobjectType: "Account",
    fieldlist: ["Id", "Name", "Industry", "Phone"],
```

```
    cacheMode: function(method) {
        if (app.offlineTracker.get("offlineStatus") == "offline") {
            return "cache-only";
        }
        else {
            return (method == "read" ?
                "cache-first" : "server-first");
        }
    }
});
```

Notice that the `app.models.Account` model object extends `Force.SObject`, which is defined in `mobilesync.js`. Also, the `cacheMode()` function queries a local `offlineTracker` object for the device's offline status. You can use the Cordova library to determine offline status at any particular moment.

Mobile Sync can perform a fetch or a save operation on the model. It uses the app's `cacheMode` value to determine whether to perform an operation on the server or in the cache. Your `cacheMode` member can either be a simple string property or a function returning a string.

Implementing a Model Collection

The model collection for this sample app extends `Force.SObjectCollection`.

```
// The AccountCollection Model
app.models.AccountCollection = Force.SObjectCollection.extend({
    model: app.models.Account,
    fieldlist: ["Id", "Name", "Industry", "Phone"],
    setCriteria: function(key) {
        this.key = key;
    },
    config: function() {
        // Offline: do a cache query
        if (app.offlineTracker.get("offlineStatus") == "offline") {
            return {type:"cache", cacheQuery:{queryType:"like",
                indexPath:"Name", likeKey: this.key+"%",
                order:"ascending"}};
        }
        // Online
        else {
            // First time: do a MRU query
            if (this.key == null) {
                return {type:"mru", sobjectType:"Account",
                    fieldlist: this.fieldlist};
            }
            // Other times: do a SOQL query
            else {
                var soql = "SELECT " + this.fieldlist.join(",")
                    + " FROM Account"
                    + " WHERE Name like '" + this.key + "%'";
                return {type:"soql", query:soql};
            }
        }
    }
});
```

This model collection uses an optional key that is the name of the account to be fetched from the collection. It also defines a `config()` function that determines what information is fetched. If the device is offline, the `config()` function builds a cache query statement. Otherwise, if no key is specified, it queries the most recently used record ("mru"). If the key is specified and the device is online, it builds a standard SOQL query that pulls records for which the name matches the key. The fetch operation on the `Force.SObjectCollection` prototype transparently uses the returned configuration to automatically fill the model collection with query records.

See querySpec for information on formatting a cache query.

Note: These code examples are part of the Account Editor sample app. See Account Editor Sample for a sample description.

Offline Caching

To provide offline support, your app must be able to cache its models and collections. Mobile Sync provides a configurable mechanism that gives you full control over caching operations.

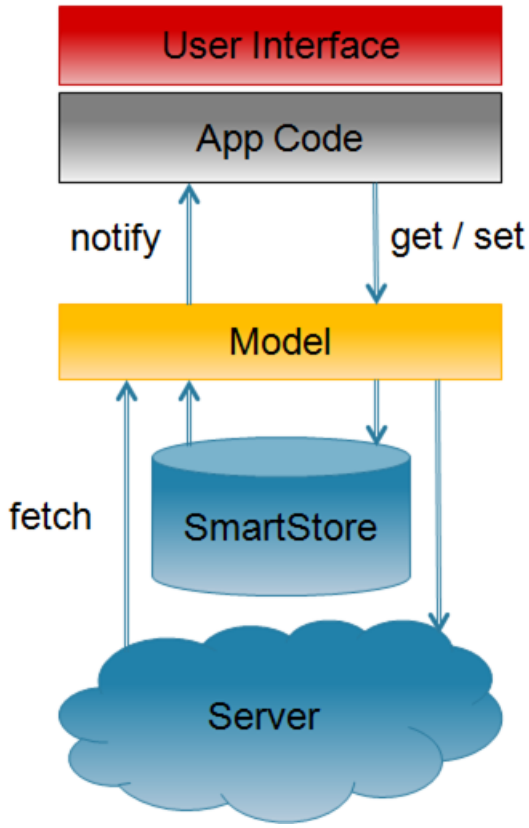
Default Cache and Custom Cache Implementations

For its default cache, the Mobile Sync library defines StoreCache, a cache implementation that uses SmartStore. Both StoreCache and SmartStore are optional components for Mobile Sync apps. If your application runs in a browser instead of the Mobile SDK container, or if you don't want to use SmartStore, you must provide an alternate cache implementation. Mobile Sync requires cache objects to support these operations:

- retrieve
- save
- save all
- remove
- find

Mobile Sync Caching Workflow

The Mobile Sync model performs all interactions with the cache and the Salesforce server on behalf of your app. Your app gets and sets attributes on model objects. During save operations, the model uses these attribute settings to determine whether to write changes to the cache or server, and how to merge new data with existing data. If anything changes in the underlying data or in the model itself, the model sends event notifications. Similarly, if you request a fetch, the model fetches the data and presents it to your app in a model collection.



Mobile Sync updates data in the cache transparently during CRUD operations. You can control the transparency level through optional flags. Cached objects maintain "dirty" attributes that indicate whether they've been created, updated, or deleted locally.

Cache Modes

When you use a cache, you can specify a mode for each CRUD operation. Supported modes are:

Mode	Constant	Description
"cache-only"	`Force.CACHE_MODE.CACHE_ONLY`	Read from, or write to, the cache. Do not perform the operation on the server.
"server-only"	`Force.CACHE_MODE.SERVER_ONLY`	Read from, or write to, the server. Do not perform the operation on the cache.
"cache-first"	`Force.CACHE_MODE.CACHE_FIRST`	For FETCH operations only. Fetch the record from the cache. If the cache doesn't contain the record, fetch it from the server and then update the cache.
"server-first" (default)	`Force.CACHE_MODE.SERVER_FIRST`	Perform the operation on the server, then update the cache.

To query the cache directly, use a cache query. SmartStore provides query APIs as well as its own query language, Smart SQL. See Retrieving Data from a Soup.

Implementing Offline Caching

To support offline caching, Mobile Sync requires you to supply your own implementations of a few tasks:

- Tracking offline status and specifying the appropriate cache control flag for CRUD operations, as shown in the `app.models.Account` example.
- Collecting records that were edited locally and saving their changes to the server when the device is back online. The following example uses a SmartStore cache query to retrieve locally changed records, then calls the `SyncPage` function to render the results in HTML.

```
sync: function() {
 var that = this;
 var localAccounts = new app.models.AccountCollection();
 localAccounts.fetch({
  config: {type:"cache", cacheQuery: {queryType:"exact",
       indexPath:"__local__", matchKey:true}},
  success: function(data) {
   that.slidePage(new app.views.SyncPage({model: data}).render());
  }
 });
}

app.views.SyncPage = Backbone.View.extend({

    template: _.template($("#sync-page").html()),

    render: function(eventName) {
        $(this.el).html(this.template(_.extend(
            {countLocallyModified: this.model.length},
            this.model.toJSON())));
        this.listView = new app.views.AccountListView(
            {el: $("ul",  this.el), model: this.model});
        this.listView.render();
        return this;
    },
...
});
```

Using StoreCache For Offline Caching

The `mobilesync.js` library implements a cache named StoreCache that stores its data in SmartStore. Although Mobile Sync uses StoreCache as its default cache, StoreCache is a stand-alone component. Even if you don't use Mobile Sync, you can still leverage StoreCache for SmartStore operations.

Note: Although StoreCache is intended for use with Mobile Sync, you can use any cache mechanism with Mobile Sync that meets the requirements described in Offline Caching.

Construction and Initialization

StoreCache objects work internally with SmartStore soups. To create a StoreCache object backed by the soup `soupName`, use the following constructor:

```
new Force.StoreCache(soupName [, additionalIndexSpecs, keyField])
```

soupName
Required. The name of the underlying SmartStore soup.

additionalIndexSpecs
Fields to include in the cache index in addition to default index fields. See Registering a Soup for formatting instructions.

keyField
Name of field containing the record ID. If not specified, StoreCache expects to find the ID in a field named "Id."

Soup items in a StoreCache object include four additional boolean fields for tracking offline edits:

- `__locally_created__`
- `__locally_updated__`
- `__locally_deleted__`
- `__local__` (set to true if any of the previous three are true)

These fields are for internal use but can also be used by apps. If your app uses the Mobile Sync plugin to sync up to the server, you're probably required to create these fields in the source soup. See Preparing Soups for Mobile Sync for instructions.

StoreCache indexes each soup on the `__local__` field and its ID field. You can use the `additionalIndexSpecs` parameter to specify additional fields to include in the index.

To register the underlying soup, call `init()` on the StoreCache object. This function returns a jQuery promise that resolves once soup registration is complete.

StoreCache Methods

init()

Registers the underlying SmartStore soup. Returns a jQuery promise that resolves when soup registration is complete.

retrieve(key [, fieldlist])

Returns a jQuery promise that resolves to the record with key in the keyField returned by the SmartStore. The promise resolves to null when no record is found or when the found record does not include all the fields in the fieldlist parameter.

key
The key value of the record to be retrieved.

fieldlist
(Optional) A JavaScript array of required fields. For example:

```
["field1","field2","field3"]
```

save(record [, noMerge])

Returns a jQuery promise that resolves to the saved record once the SmartStore upsert completes. If `noMerge` is not specified or is false, the passed record is merged with the server record with the same key, if one exists.

record

The record to be saved, formatted as:

```
{<field_name1>:"<field_value1>"[,<field_name2>:"<field_value2>",...]}
```

For example:

```
{Id:"007", Name:"JamesBond", Mission:"TopSecret"}
```

noMerge
(Optional) Boolean value indicating whether the passed record is to be merged with the matching server record. Defaults to false.

saveAll(records [, noMerge])

Identical to `save()`, except that `records` is an array of records to be saved. Returns a jQuery promise that resolves to the saved records.

records
An array of records. Each item in the array is formatted as demonstrated for the `save()` function.

noMerge
(Optional) Boolean value indicating whether the passed record is to be merged with the matching server record. Defaults to false.

remove(key)

Returns a jQuery promise that resolves when the record with the given key has been removed from the SmartStore.

key
Key value of the record to be removed.

find(querySpec)

Returns a jQuery promise that resolves once the query has been run against the SmartStore. The resolved value is an object with the following fields:

Field	Description
`records`	All fetched records
`hasMore`	Function to check if more records can be retrieved
`getMore`	Function to fetch more records
`closeCursor`	Function to close the open cursor and disable further fetch

querySpec

A specification based on SmartStore query function calls, formatted as:

```
{queryType: "like" | "exact" | "range" | "smart"[, query_type_params]}
```

where `query_type_params` match the format of the related SmartStore query function call. See Retrieving Data from a Soup.

Here are some examples:

```
{queryType:"exact", indexPath:"<indexed_field_to_match_on>", matchKey:<value_to_match>,
 order:"ascending"|"descending", pageSize:<entries_per_page>}

{queryType:"range", indexPath:"<indexed_field_to_match_on>", beginKey:<start_of_Range>,
 endKey:<end_of_range>, order:"ascending"|"descending", pageSize:<entries_per_page>}
```

```
{queryType:"like", indexPath:"<indexed_field_to_match_on>", likeKey:"<value_to_match>",
 order:"ascending"|"descending", pageSize:<entries_per_page>}

{queryType:"smart", smartSql:"<smart_sql_query>", order:"ascending"|"descending",
pageSize:<entries_per_page>}
```

Examples

The following example shows how to create, initialize, and use a StoreCache object.

```
var cache = new Force.StoreCache("agents", [{path:"Mission", type:"string"} ]);
// initialization of the cache / underlying soup
cache.init()
.then(function() {
    // saving a record to the cache
    return cache.save({Id:"007", Name:"JamesBond", Mission:"TopSecret"});
})
.then(function(savedRecord) {
    // retrieving a record from the cache
    return cache.retrieve("007");
})
.then(function(retrievedRecord) {
    // searching for records in the cache
    return cache.find({queryType:"like", indexPath:"Mission", likeKey:"Top%",
order:"ascending", pageSize:1});
})
.then(function(result) {
    // removing a record from the cache
    return cache.remove("007");
});
```

The next example shows how to use the `saveAll()` function and the results of the `find()` function.

```
// initialization
var cache = new Force.StoreCache("agents", [ {path:"Name", type:"string"}, {path:"Mission",
 type:"string"} ]);
cache.init()
.then(function() {
    // saving some records
    return cache.saveAll([{Id:"007", Name:"JamesBond"},{Id:"008", Name:"Agent008"},
{Id:"009", Name:"JamesOther"}]);
})
.then(function() {
    // doing an exact query
    return cache.find({queryType:"exact", indexPath:"Name", matchKey:"Agent008",
order:"ascending", pageSize:1});
})
.then(function(result) {
    alert("Agent mission is:" + result.records[0]["Mission"];
});
```

Conflict Detection

Model objects support optional conflict detection to prevent unwanted data loss when the object is saved to the server. You can use conflict detection with any save operation, regardless of whether the device is returning from an offline state.

To support conflict detection, you specify a secondary cache to contain the original values fetched from the server. Mobile Sync keeps this cache for later reference. When you save or delete, you specify a *merge mode*. The following table summarizes the supported modes. To understand the mode descriptions, consider "theirs" to be the current server record, "yours" the current local record, and "base" the record that was originally fetched from the server.

Mode Constant	Description
`Force.MERGE_MODE.OVERWRITE`	Write "yours" to the server, without comparing to "theirs" or "base". (This is the same as not using conflict detection.)
`Force.MERGE_MODE.MERGE_ACCEPT_YOURS`	Merge "theirs" and "yours". If the same field is changed both locally and remotely, the local value is kept.
`Force.MERGE_MODE.MERGE_FAIL_IF_CONFLICT`	Merge "theirs" and "yours". If the same field is changed both locally and remotely, the operation fails.
`Force.MERGE_MODE.MERGE_FAIL_IF_CHANGED`	Merge "theirs" and "yours". If any field is changed remotely, the operation fails.

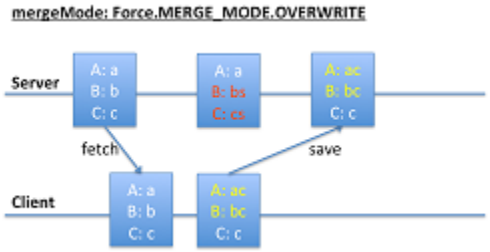
If a save or delete operation fails, you receive a report object with the following fields:

Field Name	Contains
`base`	Originally fetched attributes
`theirs`	Latest server attributes
`yours`	Locally modified attributes
`remoteChanges`	List of fields changed between base and theirs
`localChanges`	List of fields changed between base and yours
`conflictingChanges`	List of fields changed both in theirs and yours, with different values

Diagrams can help clarify how merge modes operate.

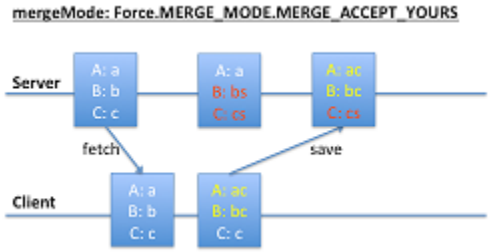
MERGE_MODE.OVERWRITE

In the `MERGE_MODE.OVERWRITE` diagram, the client changes A and B, and the server changes B and C. Changes to B conflict, whereas changes to A and C do not. However, the save operation blindly writes all the client's values to the server, overwriting any changes on the server.



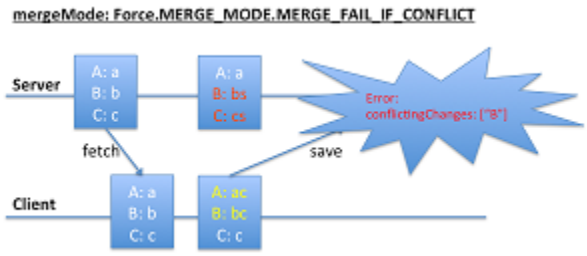
MERGE_ACCEPT_YOURS

In the `MERGE_MODE.MERGE_ACCEPT_YOURS` diagram, the client changes A and B, and the server changes B and C. Client changes (A and B) overwrites corresponding fields on the server, regardless of whether conflicts exist. However, fields that the client leaves unchanged (C) do not overwrite corresponding server values.



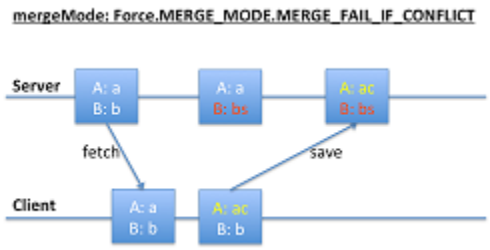
MERGE_FAIL_IF_CONFLICT (Fails)

In the first `MERGE_MODE.MERGE_FAIL_IF_CONFLICT` diagram, both the client and the server change B. These conflicting changes cause the save operation to fail.



MERGE_FAIL_IF_CONFLICT (Succeeds)

In the second `MERGE_MODE.MERGE_FAIL_IF_CONFLICT` diagram, the client changed A, and the server changed B. These changes don't conflict, so the save operation succeeds.



Mini-Tutorial: Conflict Detection

The following mini-tutorial demonstrates how merge modes affect save operations under various circumstances. It takes the form of an extended example within an HTML context.

1. Set up the necessary caches:

```
var cache = new Force.StoreCache(soupName);
var cacheForOriginals =
    new Force.StoreCache(soupNameForOriginals);
var Account = Force.SObject.extend({
    sobjectType:"Account",
    fieldlist:["Id", "Name", "Industry"],
    cache:cache,
    cacheForOriginals:cacheForOriginals});
```

2. Get an existing account:

```
var account = new Account({Id:<some actual account id>});
account.fetch();
```

3. Let's assume that the account has Name:"Acme" and Industry:"Software". Change the name to “Acme2.”

```
Account.set("Name", "Acme2");
```

4. Save to the server without specifying a merge mode, so that the default "overwrite" merge mode is used:

```
account.save(null);
```

   The account’s Name is now "Acme2" and its Industry is "Software" Let's assume that Industry changes on the server to "Electronics."

5. Change the account Name again:

```
Account.set("Name", "Acme3");
```

   You now have a change in the cache (Name) and a change on the server (Industry).

6. Save again, using "merge-fail-if-changed" merge mode.

```
account.save(null,
   {mergeMode: "merge-fail-if-changed", error: function(err) {
   // err will be a map of the form:
   // {base:…, theirs:…, yours:…,
   // remoteChanges:["Industry"], localChanges:["Name"],
   // conflictingChanges:[]}
});
```

   The error callback is called because the server record has changed.

7. Save again, using "merge-fail-if-conflict" merge mode. This merge succeeds because no conflict exists between the change on the server and the change on the client.

```
account.save(null, {mergeMode: "merge-fail-if-conflict"});
```

   The account's Name is now "Acme3" (yours) and its Industry is "Electronics" (theirs). Let's assume that, meanwhile, Name on the server changes to "NewAcme" and Industry changes to "Services."

8. Change the account Name again:

```
Account.set("Name", "Acme4");
```

9. Save again, using "merge-fail-if-changed" merge mode. The error callback is called because the server record has changed.

```
account.save(null, {mergeMode: "merge-fail-if-changed", error: function(err) {
   // err will be a map of the form:
   // {base:…, theirs:…, yours:…,
   // remoteChanges:["Name", "Industry"],
   // localChanges:["Name"], conflictingChanges:["Name"]}
});
```

10. Save again, using "merge-fail-if-conflict" merge mode:

```
account.save(null, {mergeMode: "merge-fail-if-changed", error: function(err) {
   // err will be a map of the form:
   // {base:…, theirs:…, yours:…,
   // remoteChanges:["Name", "Industry"],
   // localChanges:["Name"], conflictingChanges:["Name"]}
});
```

    The error callback is called because both the server and the cache change the Name field, resulting in a conflict:

11. Save again, using "merge-accept-yours" merge mode. This merge succeeds because your merge mode tells the `save()` function which Name value to accept. Also, since you haven't changed Industry, that field doesn't conflict.

```
account.save(null, {mergeMode: "merge-accept-yours"});
```

    Name is "Acme4" (yours) and Industry is "Services" (theirs), both in the cache and on the server.

Accessing Custom API Endpoints

In Mobile SDK 2.1, Mobile Sync expands its scope to let you work with any REST API. Previously, you could only perform basic operations on sObjects with the Salesforce Platform API. Now you can use Mobile Sync with Apex REST objects, Chatter Files, and any other Salesforce REST API. You can also call non-Salesforce REST APIs.

`Force.RemoteObject` Class

To support arbitrary REST calls, Mobile Sync introduces the `Force.RemoteObject` abstract class. `Force.RemoteObject` serves as a layer of abstraction between `Force.SObject` and `Backbone.Model`. Instead of directly subclassing `Backbone.Model`, `Force.SObject` now subclasses `Force.RemoteObject`, which in turn subclasses `Backbone.Model`. `Force.RemoteObject` does everything `Force.SObject` formerly did except communicate with the server.

Calling Custom Endpoints with `syncRemoteObjectWithServer()`

The `RemoteObject.syncRemoteObjectWithServer()` prototype method handles server interactions. `Force.SObject` implements `syncRemoteObjectWithServer()` to use the Salesforce Platform REST API. If you want to use other server end points, create a subclass of `Force.RemoteObject` and implement `syncRemoteObjectWithServer()`. This method is called when you call `fetch()` on an object of your subclass, if the object is currently configured to fetch from the server.

Example: **Example**

The FileExplorer sample application is a Mobile Sync app that shows how to use `Force.RemoteObject`. HybridFileExplorer calls the Connect REST API to manipulate files. It defines an `app.models.File` object that extends `Force.RemoteObject`. In its implementation of `syncRemoteObjectWithServer()`, `app.models.File` calls `Force.forceJsClient.fileDetails()`, which wraps the `/chatter/files/fileId` REST API.

```
app.models.File = Force.RemoteObject.extend({
    syncRemoteObjectWithServer: function(method, id) {
        if (method != "read")
            throw "Method not supported " + method;
        return Force.forceJsClient.fileDetails(id, null);
    }
})
```

Force.RemoteObjectCollection Class

To support collections of fetched objects, Mobile Sync introduces the `Force.RemoteObjectCollection` abstract class. This class serves as a layer of abstraction between `Force.SObjectCollection` and Backbone.Collection. Instead of directly subclassing Backbone.Collection, Force.SObjectCollection now subclasses Force.RemoteObjectCollection, which in turn subclasses `Backbone.Collection`. `Force.RemoteObjectCollection` does everything `Force.SObjectCollection` formerly did except communicate with the server.

Implementing Custom Endpoints with fetchRemoteObjectFromServer()

The `RemoteObject.fetchRemoteObjectFromServer()` prototype method handles server interactions. This method uses the REST API to run SOQL/SOSL and MRU queries. If you want to use arbitrary server end points, create a subclass of `Force.RemoteObjectCollection` and implement `fetchRemoteObjectFromServer()`. This method is called when you call `fetch()` on an object of your subclass, if the object is currently configured to fetch from the server.

When the `app.models.FileCollection.fetchRemoteObjectsFromServer()` function returns, it promises an object containing valuable information and useful functions that use metadata from the response. This object includes:

- `totalSize`: The number of files in the returned collection
- `records`: The collection of returned files
- `hasMore`: A function that returns a boolean value that indicates whether you can retrieve another page of results
- `getMore`: A function that retrieves the next page of results (if `hasMore()` returns true)
- `closeCursor`: A function that indicates that you're finished iterating through the collection

These functions leverage information contained in the server response, including `Files.length` and `nextPageUrl`.

Example: **Example**

The HybridFileExplorer sample application also demonstrates how to use `Force.RemoteObjectCollection`. This example calls the Connect REST API to iterate over a list of files. It supports three REST operations: `ownedFilesList`, `filesInUsersGroups`, and `filesSharedWithUser`.

You can write functions such as `hasMore()` and `getMore()`, shown in this example, to navigate through pages of results. However, since apps don't call `fetchRemoteObjectsFromServer()` directly, you capture the returned promise object when you call `fetch()` on your collection object.

```
app.models.FileCollection = Force.RemoteObjectCollection.extend({
    model: app.models.File,

    setCriteria: function(key) {
        this.config = {type:key};
    },

    fetchRemoteObjectsFromServer: function(config) {
        var fetchPromise;
        switch(config.type) {
            case "ownedFilesList": fetchPromise =
                Force.forceJsClient.ownedFilesList("me", 0);
                break;
            case "filesInUsersGroups": fetchPromise =
                Force.forceJsClient.
                    filesInUsersGroups("me", 0);
                break;
            case "filesSharedWithUser": fetchPromise =
                Force.forceJsClient.
                    filesSharedWithUser("me", 0);
                break;
        };

        return fetchPromise
            .then(function(resp) {
                var nextPageUrl = resp.nextPageUrl;
                return {
                    totalSize: resp.files.length,
                    records: resp.files,
                    hasMore: function() {
                        return nextPageUrl != null; },
                    getMore: function() {
                        var that = this;
                        if (!nextPageUrl)
                            return null;
                        return
                            forceJsClient.queryMore(nextPageUrl)
                            .then(function(resp) {
                                nextPageUrl = resp.nextPageUrl;
                                that.records.
                                    pushObjects(resp.files);
                                return resp.files;
                            });
                    },
                    closeCursor: function() {
                        return $.when(function() {
                                        nextPageUrl = null;
                                    });
                    }
                };
```

```
            });
    }
});
```

Using Apex REST Resources

To support Apex REST resources, Mobile SDK provides two classes: `Force.ApexRestObject` and `Force.ApexRestObjectCollection`. These classes subclass `Force.RemoteObject` and `Force.RemoteObjectCollection`, respectively, and can talk to a REST API that you have created using Apex REST.

`Force.ApexRestObject`

`Force.ApexRestObject` is similar to `Force.SObject`. Instead of an `sobjectType`, `Force.ApexRestObject` requires the Apex REST resource path relative to `services/apexrest`. For example, if your full resource path is `services/apexrest/simpleAccount/*`, you specify only `/simpleAccount/*`. `Force.ApexRestObject` also expects you to specify the name of your ID field if it's different from "Id".

Example: **Example**

Let's assume you've created an Apex REST resource called "simple account," which is just an account with two fields: `accountId` and `accountName`.

```
@RestResource(urlMapping='/simpleAccount/*')
  global with sharing class SimpleAccountResource {
      static String getIdFromURI() {
          RestRequest req = RestContext.request;
          return req.requestURI.substring(req.requestURI.lastIndexOf('/')+1);
      }

      @HttpGet global static Map&lt;String, String&gt; doGet() {
          String id = getIdFromURI();
          Account acc = [select Id, Name from Account
                         where Id = :id];
          return new Map&lt;String, String&gt;{
              'accountId'=>acc.Id, 'accountName'=>acc.Name};
      }

      @HttpPost global static Map&lt;String, String&gt;
          doPost(String accountName) {
              Account acc = new Account(Name=accountName);
              insert acc;
              return new Map&lt;String, String&gt;{
                  'accountId'=>acc.Id, 'accountName'=>acc.Name};
      }

      @HttpPatch global static Map&lt;String, String&gt;
          doPatch(String accountName) {
              String id = getIdFromURI();
              Account acc = [select Id from Account
                             where Id = :id];
              acc.Name = accountName;
              update acc;
              return new Map&lt;String, String&gt;{
```

```
                    'accountId'=>acc.Id, 'accountName'=>acc.Name};
        }

        @HttpDelete global static void doDelete() {
            String id = getIdFromURI();
            Account acc = [select Id from Account where Id = :id];
            delete acc;
            RestContext.response.statusCode = 204;
        }
    }
```

With Mobile Sync, you do the following to create a "simple account".

```
var SimpleAccount = Force.ApexRestObject.extend(
    {apexRestPath:"/simpleAccount",
      idAttribute:"accountId",
        fieldlist:["accountId", "accountName"]});
var acc = new SimpleAccount({accountName:"MyFirstAccount"});
acc.save();
```

You can update that "simple account".

```
acc.set("accountName", "MyFirstAccountUpdated");
acc.save(null, {fieldlist:["accountName"]);
// our apex patch endpoint only expects accountName
```

You can fetch another "simple account".

```
var acc2 = new SimpleAccount({accountId:"&lt;valid id&gt;"})
acc.fetch();
```

You can delete a "simple account".

```
acc.destroy();
```

Note: In Mobile Sync calls such as `fetch()`, `save()`, and `destroy()`, you typically pass an options parameter that defines success and error callback functions. For example:

```
acc.destroy({success:function(){alert("delete succeeded");}});
```

`Force.ApexRestObjectCollection`

`Force.ApexRestObjectCollection` is similar to `Force.SObjectCollection`. The config you specify for fetching doesn't support SOQL, SOSL, or MRU. Instead, it expects the Apex REST resource path, relative to `services/apexrest`. For example, if your full resource path is `services/apexrest/simpleAccount/*`, you specify only `/simpleAccount/*`.

You can also pass parameters for the query string if your endpoint supports them. The Apex REST endpoint is expected to return a response in this format:

```
{   totalSize: <number of records returned>
   records: <all fetched records>
   nextRecordsUrl: <url to get next records or null>
}
```

Example: **Example**

Let's assume you've created an Apex REST resource called "simple accounts". It returns "simple accounts" that match a given name.

```
@RestResource(urlMapping='/simpleAccounts/*')
global with sharing class SimpleAccountsResource {
    @HttpGet global static SimpleAccountsList doGet() {
        String namePattern =
            RestContext.request.params.get('namePattern');
        List<SimpleAccount> records = new List<SimpleAccount>();
        for (SObject sobj : Database.query(
            'select Id, Name from Account
             where Name like \'' + namePattern + '\'')) {
                Account acc = (Account) sobj;
           records.add(new
                     SimpleAccount(acc.Id, acc.Name));
        }
        return new SimpleAccountsList(records.size(), records);
    }

    global class SimpleAccountsList {
        global Integer totalSize;
        global List<SimpleAccount> records;

        global SimpleAccountsList(Integer totalSize,
            List<SimpleAccount> records) {
                this.totalSize = totalSize;
                this.records = records;
        }
    }

    global class SimpleAccount {
        global String accountId;
        global String accountName;

        global SimpleAccount(String accountId, String accountName)
        {
            this.accountId = accountId;
            this.accountName = accountName;
        }
    }
}
```

With Mobile Sync, you do the following to fetch a list of "simple account" records.

```
var getSimple = function() {
    console.log("## Trying fetch with apex rest end point");
    var config = {
        apexRestPath:"/simpleAccounts",
        params:{namePattern:accountNamePrefix + "%"}
    }
    return Force.fetchApexRestObjectsFromServer(config);
}
```

Tutorial: Creating a Hybrid Mobile Sync Application

This tutorial demonstrates how to create a local hybrid app that uses Mobile Sync. It recreates the UserSearch sample application that ships with Mobile SDK. UserSearch lets you search for User records in a Salesforce organization and see basic details about them.

This sample uses the following web technologies:

- Backbone.js
- Ratchet
- HTML5
- JavaScript

Create a Template Project

First, make sure you've installed Salesforce Mobile SDK using the NPM installer. For iOS instructions, see iOS Preparation. For Android instructions, see Android Preparation.

Also, download the `ratchet.css` file from http://goratchet.com/.

Once you've installed Mobile SDK, create a local hybrid project for your platform.

1. At a Terminal window or Windows command prompt, run the *`forcehybrid create`* command using the following values:

Prompt (or Parameter)	Value
Platform (`--platform`)	*`ios`*, *`android`*, or *`ios,android`*
Application type (`--apptype`)	*`hybrid_local`*
Application name (`--appname`)	*`UserSearch`*
Package name (`--packagename`)	*`com.acme.usersearch`*
Organization (`--organization`)	*`"Acme Widgets, Inc."`*
Output directory (`--outputdir`)	Leave blank for current directory, or enter a name to create a new subdirectory for the project

   Here's a command line example:

```
forcehybrid create --platform=ios,android --apptype=hybrid_local
    --appname=UserSearch --packagename=com.acme.usersearch
    --organization="Acme Widgets, Inc." --outputdir=""
```

2. Copy all files—actual and symbolic—from the `samples/usersearch` directory of the https://github.com/forcedotcom/SalesforceMobileSDK-Shared/ repository into the `www/` folder, as follows:
   - In a Mac OS X terminal window, change to your project's root directory—./UserSearch/—and type this command:

     ```
     cp -RL <insert local path to SalesforceMobileSDK-Shared>/samples/UserSearch/* www/
     ```

   - In Windows, make sure that every file referenced in the *`<shared repo>`*`\samples\usersearch` folder also appears in your *`<project_name>`*`\www` folder. Resolve the symbolic links explicitly, as shown in the following script:

     ```
     cd <your project's root directory>
     set SHARED_REPO=<insert local path to SalesforceMobileSDK-Shared>
     ```

```
copy %SHARED_REPO%\samples\usersearch\UserSearch.html www
copy %SHARED_REPO%\samples\usersearch\bootconfig.json www
copy %SHARED_REPO%\dependencies\ratchet\ratchet.css www
copy %SHARED_REPO%\samples\common\styles.css www
copy %SHARED_REPO%\test\MockCordova.js www
copy %SHARED_REPO%\samples\common\auth.js www
copy %SHARED_REPO%\dependencies\backbone\backbone-min.js www
copy %SHARED_REPO%\libs\cordova.force.js www
copy %SHARED_REPO%\dependencies\fastclick\fastclick.js www
copy %SHARED_REPO%\libs\force.js www
copy %SHARED_REPO%\libs\force+promise.js www
copy %SHARED_REPO%\dependencies\jquery\jquery.min.js www
copy %SHARED_REPO%\libs\mobilesync.js www
copy %SHARED_REPO%\samples\common\stackrouter.js www
copy %SHARED_REPO%\dependencies\underscore\underscore-min.js www
```

3. Run the following command:

```
cordova prepare
```

4. Open the `platforms/android/` project folder in Android Studio (for Android) or Xcode (for iOS) by following the onscreen instructions printed by forcehybrid.
5. From the `www` folder, open `UserSearch.html` in your code editor and delete all its contents.

Edit the Application HTML File

To create your app's basic structure, define an empty HTML page that contains references, links, and code infrastructure.

1. From the `www` folder, open `UserSearch.html` in your code editor and delete all its contents.
2. Delete the contents and add the following basic structure:

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
  </body>
</html>
```

3. In the `<head>` element:
   a. Specify that the page title is "Users".

```
<title>Users</title>
```

   b. Turn off scaling to make the page look like an app rather than a web page.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0,
    maximum-scale=1.0, user-scalable=no;" />
```

   c. Provide a mobile "look" by adding links to the `styles.css` and `ratchet.css` files.

```
<link rel="stylesheet" href="css/styles.css"/>
<link rel="stylesheet" href="css/ratchet.css"/>
```

4. Now let's start adding content to the body. In the `<body>` block, add an empty `div` tag, with ID set to "content", to contain the app's generated UI.

```
<body>
<div id="content"></div>
```

5. Include the necessary JavaScript files.

```
<script src="js/jquery.min.js"></script>
<script src="js/underscore-min.js"></script>
<script src="js/backbone-min.js"></script>
<script src="cordova.js"></script>
<script src="js/force.js"></script>
<script src="js/force+promise.js"></script>
<script src="js/mobilesync.js"></script>
<script src="js/fastclick.js"></script>
<script src="js/stackrouter.js"></script>
<script src="js/auth.js"></script>
```

Example: Here's the complete application to this point.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Users</title>
    <meta name="viewport" content="width=device-width,
      initial-scale=1.0, maximum-scale=1.0;
      user-scalable=no" />
    <link rel="stylesheet" href="css/styles.css"/>
    <link rel="stylesheet" href="css/ratchet.css"/>
  </head>
  <body>
    <div id="content"></div>
    <script src="js/jquery.min.js"></script>
    <script src="js/underscore-min.js"></script>
    <script src="js/backbone-min.js"></script>
    <script src="cordova.js"></script>
    <script src="js/force.js"></script>
    <script src="js/force+promise.js"></script>
    <script src="js/mobilesync.js"></script>
    <script src="js/fastclick.js"></script>
    <script src="js/stackrouter.js"></script>
    <script src="js/auth.js"></script>
  </body>
</html>
```

Create a Mobile Sync Model and a Collection

Now that we've configured the HTML infrastructure, let's get started using Mobile Sync by extending two of its primary objects:

- `Force.SObject`
- `Force.SObjectCollection`

These objects extend `Backbone.Model`, so they support the `Backbone.Model.extend()` function. To extend an object using this function, pass it a JavaScript object containing your custom properties and functions.

1. In the `<body>` tag, create a `<script>` object.
2. In the `<script>` tag, create a model object for the Salesforce user sObject. Extend `Force.SObject`, and specify the sObject type and the fields we are targeting.

```
app.models.User = Force.SObject.extend({
    sobjectType: "User",
    fieldlist: ["Id", "FirstName", "LastName",
        "SmallPhotoUrl", "Title", "Email",
        "MobilePhone","City"]
})
```

3. Immediately after setting the User object, create a `UserCollection` object to hold user search results. Extend `Force.SObjectCollection`, and specify your new model (`app.models.User`) as the model for items in the collection.

```
app.models.UserCollection = Force.SObjectCollection.extend({
    model: app.models.User,
    fieldlist: ["Id", "FirstName", "LastName",
        "SmallPhotoUrl", "Title"],

});
```

4. In this collection, implement a function named `setCriteria` that takes a search key and builds a SOQL query using it. You also need a getter to return the key at a later point.

```
<script>
    // The Models
    // ==========
    // The User Model
    app.models.User = Force.SObject.extend({
        sobjectType: "User",
        fieldlist: ["Id", "FirstName",
            "LastName", "SmallPhotoUrl",
            "Title", "Email",
            "MobilePhone","City"]
    });

    // The UserCollection Model
    app.models.UserCollection = Force.SObjectCollection.extend({
        model: app.models.User
        fieldlist: ["Id", "FirstName", "LastName",
            "SmallPhotoUrl", "Title"],

        getCriteria: function() {
            return this.key;
        },

        setCriteria: function(key) {
            this.key = key;
            this.config = {type:"soql", query:"SELECT "
                + this.fieldlist.join(",")
                + " FROM User"
```

```
                + " WHERE Name like '" + key + "%'"
                + " ORDER BY Name "
                + " LIMIT 25 "
            };
        }
    });
</script>
```

Example: Here's the complete model code.

```
<script>
    // The Models

    // The User Model
    app.models.User = Force.SObject.extend({
        sobjectType: "User",
        fieldlist: ["Id", "FirstName", "LastName",
            "SmallPhotoUrl", "Title", "Email",
            "MobilePhone","City"]
    });

    // The UserCollection Model
    app.models.UserCollection = Force.SObjectCollection.extend({
        model: app.models.User
        fieldlist: ["Id", "FirstName", "LastName",
            "SmallPhotoUrl", "Title"],

        getCriteria: function() {
            return this.key;
        },

        setCriteria: function(key) {
            this.key = key;
            this.config = {
                type:"soql",
                query:"SELECT " + this.fieldlist.join(",")
                + " FROM User"
                + " WHERE Name like '" + key + "%'"
                + " ORDER BY Name "
                + " LIMIT 25 "
             };
        }
    });
</script>
```

Create View Templates

Templates let you describe an HTML layout within a container HTML page. To define an inline template in your HTML page, you use a `<script>` tag of type "text/template". JavaScript code can apply your template to the page design when it instantiates a new HTML page at runtime.

The `search-page` template is simple. It includes a header, a search field, and a list to hold the search results. At runtime, the search page instantiates the `user-list-item` template to render the results list. When a customer clicks a list item, the list instantiates the `user-page` template to show user details.

1. Add a template script block with an ID set to "search-page". Place the block within the `<body>` block after the "content" `<div>` tag.

```
<script id="search-page" type="text/template">
</script>
```

2. In the new `<script>` block, define the search page HTML template using Ratchet styles.

```
<script id="search-page" type="text/template">
  <header class="bar-title">
    <h1 class="title">Users</h1>
  </header>

  <div class="bar-standard bar-header-secondary">
    <input type="search" class="search-key"
        placeholder="Search"/>
  </div>

  <div class="content">
    <ul class="list"></ul>
  </div>
</script>
```

3. Add a second script block for a user list template.

```
<script id="user-list-item" type="text/template">
</script>
```

4. Define the user list template. Notice that this template contains references to the `SmallPhotoUrl`, `FirstName`, `LastName`, and `Title` fields from the Salesforce user record. References that use the `<%= varname %>` format are called "free variables" in Ratchet apps.

```
<script id="user-list-item" type="text/template">
  <a href="#users/<%= Id %>" class="pad-right">
    <img src="<%= SmallPhotoUrl %>" class="small-img" />
    <div class="details-short">
      <b><%= FirstName %> <%= LastName %></b><br/>
      Title<%= Title %>
    </div>
  </a>
</script>
```

5. Add a third script block for a user details template.

```
<script id="user-page" type="text/template">
</script>
```

6. Add the template body. Notice that this template contains references to the `SmallPhotoUrl`, `FirstName`, `LastName`, and `Title` fields from the Salesforce user record. References that use the `<%= varname %>` format in Ratchet apps are called "free variables".

```
<script id="user-page" type="text/template">
  <header class="bar-title">
    <a href="#" class="button-prev">Back</a>
    <h1 class="title">User</h1>
  </header>

  <footer class="bar-footer">
    <span id="offlineStatus"></span>
  </footer>

  <div class="content">
    <div class="content-padded">
      <img id="employeePic" src="<%= SmallPhotoUrl %>"
         class="large-img" />
      <div class="details">
        <b><%= FirstName %> <%= LastName %></b><br/>
        <%= Id %><br/>
        <% if (Title) { %><%= Title %><br/><% } %>
        <% if (City) { %><%= City %><br/><% } %>
        <% if (MobilePhone) { %> <a
           href="tel:<%= MobilePhone %>">
           <%= MobilePhone %></a><br/><% } %>
        <% if (Email) { %><a
           href="mailto:<%= Email %>">
           <%= Email %></a><% } %>
      </div>
    </div>
  </div>
</script>
```

Add the Search View

To create the view for a screen, you extend `Backbone.View`. Let's start by defining the search view. In this extension, you load the template, define subviews and event handlers, and implement the functionality for rendering the views and performing a SOQL search query.

1. In the `<script>` block where you defined the User and UserCollection models, create a `Backbone.View` extension named `SearchPage` in the `app.views` array.

```
app.views.SearchPage = Backbone.View.extend({
});
```

   For the remainder of this procedure, add all code to the `extend({})` block. Each step adds another item to the implementation list and therefore ends with a comma, until the last item.

2. Load the search-page template by calling the `_.template()` function. Pass it the raw HTML content of the `search-page` script tag.

```
template: _.template($("#search-page").html()),
```

3. Add a `keyup` event. You define the `search` handler function a little later.

```
events: {
    "keyup .search-key": "search"
},
```

4. Instantiate a subview named `UserListView` that contains the list of search results. (You define `app.views.UserListView` later.)

```
initialize: function() {
    this.listView = new app.views.UserListView({model: this.model});
},
```

5. Create a `render()` function for the search page view. Rendering the view consists of loading the template as the app's HTML content. Restore any criteria previously typed in the search field and render the subview inside the `<ul>` element.

```
render: function(eventName) {
    $(this.el).html(this.template());
    $(".search-key", this.el).val(this.model.getCriteria());
    this.listView.setElement($("ul", this.el)).render();
    return this;
},
```

6. Implement the `search` function. This function is the `keyup` event handler that performs a search when the customer types a character in the search field.

```
search: function(event) {
    this.model.setCriteria($(".search-key", this.el).val());
    this.model.fetch();
}
```

Example: Here's the complete extension.

```
app.views.SearchPage = Backbone.View.extend({
    template: _.template($("#search-page").html()),
    events: {
        "keyup .search-key": "search"
    },
    initialize: function() {
        this.listView = new app.views.UserListView({model: this.model});
    },
    render: function(eventName) {
        $(this.el).html(this.template());
        $(".search-key", this.el).val(this.model.getCriteria());
        this.listView.setElement($("ul", this.el)).render();
        return this;
    },
    search: function(event) {
        this.model.setCriteria($(".search-key", this.el).val());
        this.model.fetch();
```

```
    }
});
```

Add the Search Result List View

The view for the search result list doesn't need a template. It is simply a container for list item views. It tracks these views in the `listItemViews` member. If the underlying collection changes, it re-renders itself.

1. In the `<script>` block that contains the `SearchPage` view, extend `Backbone.View` to show a list of search view results. Add an array for list item views and an `initialize()` function.

   ```
   app.views.UserListView = Backbone.View.extend({
       listItemViews: [],
       initialize: function() {
           this.model.bind("reset", this.render, this);
       },
   ```

   For the remainder of this procedure, add all code to the `extend({})` block.

2. Create the `render()` function. This function cleans up any existing list item views by calling `close()` on each one.

   ```
   render: function(eventName) {
       _.each(this.listItemViews,
              function(itemView) { itemView.close(); });
   ```

3. Still in the `render()` function, create a set of list item views for the records in the underlying collection. Each of these views is just an entry in the list. You define `app.views.UserListItemView` later.

   ```
   this.listItemViews = _.map(this.model.models, function(model) { return new
       app.views.UserListItemView({model: model}); });
   ```

4. Still in the `render()` function, append each list item view to the root DOM element and then return the rendered `UserListView` object.

   ```
   $(this.el).append(_.map(this.listItemViews, function(itemView) {
       return itemView.render().el;} ));
       return this;
   }
   ```

Example: Here's the complete extension:

```
app.views.UserListView = Backbone.View.extend({

    listItemViews: [],

    initialize: function() {
        this.model.bind("reset", this.render, this);
    },
    render: function(eventName) {
        _.each(this.listItemViews, function(itemView) {
            itemView.close(); });
        this.listItemViews = _.map(this.model.models,
            function(model) {
                return new app.views.UserListItemView(
```

```
                    {model: model}); });
        $(this.el).append(_.map(this.listItemViews,
            function(itemView) {
                return itemView.render().el;
            } ));
        return this;
    }
});
```

Add the Search Result List Item View

To define the search result list item view, you design and implement the view of a single row in a list. Each list item displays the following user fields:

- SmallPhotoUrl
- FirstName
- LastName
- Title

1. Immediately after the `UserListView` view definition, create the view for the search result list item. Once again, extend `Backbone.View` and indicate that this view is a list item by defining the `tagName` member. For the remainder of this procedure, add all code in the `extend({})` block.

```
app.views.UserListItemView = Backbone.View.extend({

});
```

2. Add an `<li>` tag.

```
app.views.UserListItemView = Backbone.View.extend({
    tagName: "li",
});
```

3. Load the template by calling `_.template()` with the raw content of the `user-list-item` script.

```
template: _.template($("#user-list-item").html()),
```

4. Add a `render()` function. The `template()` function, from `underscore.js`, takes JSON data and returns HTML crafted from the associated template. In this case, the function extracts the customer's data from JSON and returns HTML that conforms to the `user-list-item` template. During the conversion to HTML, the `template()` function replaces free variables in the template with corresponding properties from the JSON data.

```
render: function(eventName) {
    $(this.el).html(this.template(this.model.toJSON()));
    return this;
},
```

5. Add a `close()` method to be called from the list view that does necessary cleanup and stops memory leaks.

```
close: function() {
    this.remove();
    this.off();
}
```

Example: Here's the complete extension.

```
app.views.UserListItemView = Backbone.View.extend({
    tagName: "li",
    template: _.template($("#user-list-item").html()),
    render: function(eventName) {
        $(this.el).html(this.template(this.model.toJSON()));
        return this;
    },
    close: function() {
        this.remove();
        this.off();
    }
});
```

Add the User View

Finally, you add a simple page view that displays a selected customer's details. This view is the second page in this app. The customer navigates to it by tapping an item in the Users list view. The `user-page` template defines a **Back** button that returns the customer to the search list.

1. Immediately after the `UserListItemView` view definition, create the view for a customer's details. Extend `Backbone.View` again. For the remainder of this procedure, add all code in the `extend({})` block.

```
app.views.UserPage = Backbone.View.extend({

});
```

2. Specify the template to be instantiated.

```
app.views.UserPage = Backbone.View.extend({
    template: _.template($("#user-page").html()),
});
```

3. Implement a `render()` function. This function re-reads the model and converts it first to JSON and then to HTML.

```
app.views.UserPage = Backbone.View.extend({
    template: _.template($("#user-page").html()),

    render: function(eventName) {
        $(this.el).html(this.template(this.model.toJSON()));
        return this;
    }
});
```

Example: Here's the complete extension.

```
app.views.UserPage = Backbone.View.extend({
    template: _.template($("#user-page").html()),
    render: function(eventName) {
        $(this.el).html(this.template(this.model.toJSON()));
        return this;
    }
});
```

Define a Router

A Backbone router defines navigation paths among views. To learn more about routers, see What is a router?

1. In the final `<script>` block, define the application router by extending `Backbone.StackRouter`.

```
app.Router = Backbone.StackRouter.extend({

});
```

   For the remainder of this procedure, add all code in the `extend({})` block.

2. Because the app supports a search list page and a user page, add a route for each page inside a `routes` object. Also add a route for the main container page ("").

```
routes: {
    "": "list",
    "list": "list",
    "users/:id": "viewUser"
},
```

3. Define an `initialize()` function that creates the search results collection and the search page and user page views.

```
initialize: function() {
    Backbone.Router.prototype.initialize.call(this);

    // Collection behind search screen
    app.searchResults = new app.models.UserCollection();

    app.searchPage = new app.views.SearchPage(
        {model: app.searchResults});
    app.userPage = new app.views.UserPage();
},
```

4. Define the `list()` function for handling the only item in this route. Call `slidePage()` to show the search results page right away—when data arrives, the list redraws itself.

```
list: function() {
   app.searchResults.fetch();
   this.slidePage(app.searchPage);
},
```

5. Define a `viewUser()` function that fetches and displays details for a specific user.

```
viewUser: function(id) {
    var that = this;
    var user = new app.models.User({Id: id});
    user.fetch({
        success: function() {
            app.userPage.model = user;
            that.slidePage(app.userPage);
        }
    });
}
```

6. After saving the file, run the *`cordova prepare`* command.

7. Run the application.

Example: You've finished! Here's the entire application:

```
<!DOCTYPE html>
<html>
<head>
<title>Users</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0,
 user-scalable=no;" />
<link rel="stylesheet" href="css/styles.css"/>
<link rel="stylesheet" href="css/ratchet.css"/>
</head>

<body>

<div id="content"></div>
<script src="js/jquery.min.js"></script>
<script src="js/underscore-min.js"></script>
<script src="js/backbone-min.js"></script>

<!-- Local Testing -/->
<script src="js/MockCordova.js"></script>
<script src="js/cordova.force.js"></script>
<script src="js/MockSmartStore.js"></script>
<!-- End Local Testing -->

<!-- Container -->
<script src="cordova.js"></script>
<!-- End Container -->

<script src="js/force.js"></script>
<script src="js/force+promise.js"></script>
<script src="js/mobilesync.js"></script>
<script src="js/fastclick.js"></script>
<script src="js/stackrouter.js"></script>
<script src="js/auth.js"></script>


<!-- ----Search page template ---- -->
<script id="search-page" type="text/template">
  <header class="bar-title">
    <h1 class="title">Users</h1>
  </header>

  <div class="bar-standard bar-header-secondary">
    <input type="search"
     class="search-key"
     placeholder="Search"/>
  </div>

  <div class="content">
    <ul class="list"></ul>
  </div>
</script>
```

```
<!-- ---- User list item template ---- -->
<script id="user-list-item" type="text/template">

  <a href="#users/<%= Id %>" class="pad-right">
    <img src="<%= SmallPhotoUrl %>" class="small-img" />
    <div class="details-short">
      <b><%= FirstName %> <%= LastName %></b><br/>
      Title<%= Title %>
    </div>
  </a>
</script>

<!-- ---- User page template ---- -->
<script id="user-page" type="text/template">
  <header class="bar-title">
    <a href="#" class="button-prev">Back</a>
    <h1 class="title">User</h1>
  </header>

  <footer class="bar-footer">
    <span id="offlineStatus"></span>
  </footer>

  <div class="content">
    <div class="content-padded">
      <img id="employeePic"
         src="<%= SmallPhotoUrl %>" class="large-img" />
      <div class="details">
        <b><%= FirstName %> <%= LastName %></b><br/>
        <%= Id %><br/>
        <% if (Title) { %><%= Title %><br/><% } %>
        <% if (City) { %><%= City %><br/><% } %>
        <% if (MobilePhone) { %>
           <a href="tel:<%= MobilePhone %>">
           <%= MobilePhone %></a><br/><% } %>
        <% if (Email) { %>
           <a href="mailto:<%= Email %>">
           <%= Email %></a><% } %>
      </div>
    </div>
  </div>
</script>

<script>
// ---- The Models ---- //
// The User Model
app.models.User = Force.SObject.extend({
    sobjectType: "User",
    fieldlist: ["Id", "FirstName", "LastName", "SmallPhotoUrl",
       "Title", "Email", "MobilePhone","City"]
});

// The UserCollection Model
```

```
app.models.UserCollection = Force.SObjectCollection.extend({
    model: app.models.User,
    fieldlist: ["Id", "FirstName", "LastName", "SmallPhotoUrl",
        "Title"],

    getCriteria: function() {
        return this.key;
    },

    setCriteria: function(key) {
        this.key = key;
        this.config = {type:"soql",
                       query:"SELECT "
                        + this.fieldlist.join(",")
                        + " FROM User"
                        + " WHERE Name like '" + key + "%'"
                        + " ORDER BY Name "
                        + " LIMIT 25 "
                      };
    }
});


// ------------------------------------------------ The Views
-------------------------------------------------- //

app.views.SearchPage = Backbone.View.extend({

    template: _.template($("#search-page").html()),

    events: {
        "keyup .search-key": "search"
    },

    initialize: function() {
        this.listView =
            new app.views.UserListView(
                {model: this.model});
    },

    render: function(eventName) {
        $(this.el).html(this.template());
        $(".search-key", this.el).val(this.model.getCriteria());
        this.listView.setElement($("ul", this.el)).render();
        return this;
    },

    search: function(event) {
        this.model.setCriteria($(".search-key", this.el).val());
        this.model.fetch();
    }
});

app.views.UserListView = Backbone.View.extend({
```

```
    listItemViews: [],

    initialize: function() {
        this.model.bind("reset", this.render, this);
    },

    render: function(eventName) {
        _.each(this.listItemViews,
            function(itemView) {itemView.close(); });
        this.listItemViews =
            _.map(this.model.models, function(model) {
                 return new app.views.UserListItemView(
                     {model: model}); });
        $(this.el).append(_.map(this.listItemViews,
            function(itemView) {
                return itemView.render().el;} ));
        return this;
    }
});

app.views.UserListItemView = Backbone.View.extend({

    tagName: "li",
    template: _.template($("#user-list-item").html()),

    render: function(eventName) {
        $(this.el).html(this.template(this.model.toJSON()));
        return this;
    },

    close: function() {
        this.remove();
        this.off();
    }
});

app.views.UserPage = Backbone.View.extend({

    template: _.template($("#user-page").html()),

    render: function(eventName) {
        $(this.el).html(this.template(this.model.toJSON()));
        return this;
    }

});


// ---------------------------------------------- The Application Router
----------------------------------------- //

app.Router = Backbone.StackRouter.extend({
```

```
    routes: {
        "": "list",
        "list": "list",
        "users/:id": "viewUser"
    },

    initialize: function() {
        Backbone.Router.prototype.initialize.call(this);

        // Collection behind search screen
        app.searchResults = new app.models.UserCollection();

        // We keep a single instance of SearchPage and UserPage
        app.searchPage = new app.views.SearchPage(
            {model: app.searchResults});
        app.userPage = new app.views.UserPage();
    },

    list: function() {
        app.searchResults.fetch();
        // Show page right away
        // List will redraw when data comes in
        this.slidePage(app.searchPage);
    },

    viewUser: function(id) {
        var that = this;
        var user = new app.models.User({Id: id});
        user.fetch({
            success: function() {
                app.userPage.model = user;
                that.slidePage(app.userPage);
            }
        });
    }
});
</script>
</body>
</html>
```

Mobile Sync Sample Apps

Salesforce Mobile SDK provides sample apps that demonstrate how to use Mobile Sync in hybrid apps. Account Editor is the most full-featured of these samples. You can switch to one of the simpler samples by changing the `startPage` property in the `bootconfig.json` file.

Running the Samples in iOS

In your Salesforce Mobile SDK for iOS installation directory, double-click the `SalesforceMobileSDK.xcworkspace` to open it in Xcode. In Xcode Project Navigator, select the `Hybrid SDK/AccountEditor` project and click **Run**.

Running the Samples in Android

To run the sample in Android Studio, you first add references to basic libraries from your clone of the SalesforceMobileSDK-Android repository. Add the following dependencies to your sample module, setting **Scope** to "Compile" for each one:

- `libs/SalesforceSDK`
- `libs/SmartStore`
- `hybrid/SampleApps/AccountEditor`

After Android Studio finishes building, click **Run '<*sample_name*>'** in the toolbar or menu.

Account Editor Sample

Account Editor is the most complex Mobile Sync-based sample application in Mobile SDK 2.0. It allows you to create/edit/update/delete accounts online and offline, and also demonstrates conflict detection.

To run the sample:

1. If you've made changes to `external/shared/sampleApps/mobilesync/bootconfig.json`, revert it to its original content.
2. Launch Account Editor.

This application contains three screens:

- Accounts search
- Accounts detail
- Sync

When the application first starts, you see the Accounts search screen listing the most recently used accounts. In this screen, you can:

- Type a search string to find accounts whose names contain the given string.
- Tap an account to launch the account detail screen.
- Tap **Create** to launch an empty account detail screen.
- Tap **Online** to go offline. If you are already offline, you can tap the **Offline** button to go back online. (You can also go offline by putting the device in airplane mode.)

To launch the Account Detail screen, tap an account record in the Accounts search screen. The detail screen shows you the fields in the selected account. In this screen, you can:

- Tap a field to change its value.
- Tap **Save** to update or create the account. If validation errors occur, the fields with problems are highlighted.

  If you're online while saving and the server's record changed since the last fetch, you receive warnings for the fields that changed remotely.

  Two additional buttons, **Merge** and **Overwrite**, let you control how the app saves your changes. If you tap **Overwrite,** the app saves to the server all values currently displayed on your screen. If you tap **Merge**, the app saves to the server only the fields you changed, while keeping changes on the server in fields you did not change.
- Tap **Delete** to delete the account.
- Tap **Online** to go offline, or tap **Offline** to go online.

To see the Sync screen, tap **Online** to go offline, then create, update, or delete an account. When you tap **Offline** again to go back online, the Sync screen shows all accounts that you modified on the device.

Tap **Process *n* records** to try to save your local changes to the server. If any account fails to save, it remains in the list with a notation that it failed to sync. You can tap any account in the list to edit it further or, in the case of a locally deleted record, to undelete it.

Looking Under the Hood

To view the source code for this sample, open `AccountEditor.html` in an HTML or text editor.

Here are the key sections of the file:

- Script includes
- Templates
- Models
- Views
- Router

Script Includes

This sample includes the standard list of libraries for Mobile Sync applications.

- jQuery—See http://jquery.com/.
- Underscore—Utility-belt library for JavaScript, required by backbone. See http://underscorejs.org/.
- Backbone—Gives structure to web applications. Used by Mobile Sync. See http://backbonejs.org/.
- `cordova.js`—Required for hybrid applications using the Salesforce Mobile SDK.
- `force.js`—Salesforce Platform JavaScript library for making REST API calls. Required by Mobile Sync.
- `mobilesync.js`—Mobile Sync.
- `fastclick.js`—Library used to eliminate the 300 ms delay between physical tap and firing of a click event. See https://github.com/ftlabs/fastclick.
- `stackrouter.js` and `auth.js`—Helper JavaScript libraries used by all three sample applications.

Templates

Templates for this application include:

- search-page
- sync-page
- account-list-item
- edit-account-page (for the Account detail page)

Models

This sample defines three models: `AccountCollection`, `Account` and `OfflineTracker`.

`AccountCollection` is a subclass of Mobile Sync's `Force.SObjectCollection` class, which is a subclass of the Backbone framework's `Collection` class.

The `AccountCollection.config()` method returns an appropriate query to the collection. The query mode can be:

- Most recently used (MRU) if you are online and haven't provided query criteria
- SOQL if you are online and have provided query criteria
- SmartSQL when you are offline

When the app calls `fetch()` on the collection, the `fetch()` function executes the query returned by `config()`. It then uses the results of this query to populate `AccountCollection` with Account objects from either the offline cache or the server.

`AccountCollection` uses the two global caches set up by the AccountEditor application: `app.cache` for offline storage, and `app.cacheForOriginals` for conflict detection. The code shows that the `AccountCollection` model:

- Contains objects of the `app.models.Account` model (`model` field)
- Specifies a list of fields to be queried (`fieldlist` field)
- Uses the sample app's global offline cache (`cache` field)
- Uses the sample app's global conflict detection cache (`cacheForOriginals` field)
- Defines a `config()` function to handle online as well as offline queries

Here's the code (shortened for readability):

```
app.models.AccountCollection = Force.SObjectCollection.extend({
    model: app.models.Account,
    fieldlist: ["Id", "Name", "Industry", "Phone", "Owner.Name",
         "LastModifiedBy.Name", "LastModifiedDate"],
    cache: function() { return app.cache},
    cacheForOriginals: function() {
        return app.cacheForOriginals;},

    config: function() {
        // Offline: do a cache query
        if (!app.offlineTracker.get("isOnline")) {
            // ...
        }
        // Online
        else {
            // ...
        }
    }
});
```

`Account` is a subclass of Mobile Sync's `Force.SObject` class, which is a subclass of the Backbone framework's `Model` class. Code for the `Account` model shows that it:

- Uses a `sobjectType` field to indicate which type of sObject it represents (Account, in this case).
- Defines `fieldlist` as a method rather than a field, because the fields that it retrieves from the server are not the same as the ones it sends to the server.
- Uses the sample app's global offline cache (`cache` field).
- Uses the sample app's global conflict detection cache (`cacheForOriginals` field).
- Supports a `cacheMode()` method that returns a value indicating how to handle caching based on the current offline status.

Here's the code:

```
app.models.Account = Force.SObject.extend({
    sobjectType: "Account",
    fieldlist: function(method) {
        return method == "read"
            ? ["Id", "Name", "Industry", "Phone", "Owner.Name",
                  "LastModifiedBy.Name", "LastModifiedDate"]
            : ["Id", "Name", "Industry", "Phone"];
    },
    cache: function() { return app.cache;},
    cacheForOriginals: function() { return app.cacheForOriginals;},
    cacheMode: function(method) {
```

```
        if (!app.offlineTracker.get("isOnline")) {
            return Force.CACHE_MODE.CACHE_ONLY;
        }
        // Online
        else {
            return (method == "read" ?
                Force.CACHE_MODE.CACHE_FIRST :
                Force.CACHE_MODE.SERVER_FIRST);
        }
    }
});
```

`OfflineTracker` is a subclass of Backbone's `Model` class. This class tracks the offline status of the application by observing the browser's offline status. It automatically switches the app to offline when it detects that the browser is offline. However, it goes online only when the user requests it.

Here's the code:

```
app.models.OfflineTracker = Backbone.Model.extend({
    initialize: function() {
        var that = this;
        this.set("isOnline", navigator.onLine);
        document.addEventListener("offline", function() {
            console.log("Received OFFLINE event");
            that.set("isOnline", false);
        }, false);
        document.addEventListener("online", function() {
            console.log("Received ONLINE event");
            // User decides when to go back online
        }, false);
    }
});
```

Views

This sample defines five views:

- SearchPage
- AccountListView
- AccountListItemView
- EditAccountView
- SyncPage

A view typically provides a template field to specify its design template, an `initialize()` function, and a `render()` function.

Each view can also define an `events` field. This field contains an array whose key/value entries specify the event type and the event handler function name. Entries use the following format:

```
"<event-type>[ <control>]": "<event-handler-function-name>"
```

For example:

```
events: {
    "click .button-prev": "goBack",
    "change": "change",
```

```
    "click .save": "save",
    "click .merge": "saveMerge",
    "click .overwrite": "saveOverwrite",
    "click .toggleDelete": "toggleDelete"
},
```

SearchPage

View for the entire search screen. It expects an `AccountCollection` as its model. It watches the search input field for changes (the `keyup` event) and updates the model accordingly in the `search()` function.

```
events: {
    "keyup .search-key": "search"
},
search: function(event) {
    this.model.setCriteria($(".search-key", this.el).val());
    this.model.fetch();
}
```

AcountListView

View for the list portion of the search screen. It expects an `AccountCollection` as its model and creates `AccountListItemView` object for each account in the `AccountCollection` object.

AccountListItemView

View for an item within the list.

EditAccountPage

View for account detail page. This view monitors several events:

Event Type	Target Control	Handler function name
click	button-prev	goBack
change	Not set (can be any edit control)	change
click	save	save
click	merge	saveMerge
click	overwrite	saveOverwrite
click	toggleDelete	toggleDelete

A couple of event handler functions deserve special attention. The `change()` function shows how the view uses the event target to send user edits back to the model:

```
change: function(evt) {
    // apply change to model
    var target = event.target;
    this.model.set(target.name, target.value);
    $("#account" + target.name + "Error", this.el).hide();
}
```

The `toggleDelete()` function handles a toggle that lets the user delete or undelete an account. If the user clicks to undelete, the code sets an internal `__locally_deleted__ flag` to false to indicate that the record is no longer deleted in the cache. Else, it attempts to delete the record on the server by destroying the local model.

```
toggleDelete: function() {
    if (this.model.get("__locally_deleted__")) {
        this.model.set("__locally_deleted__", false);
        this.model.save(null, this.getSaveOptions(
            null, Force.CACHE_MODE.CACHE_ONLY));
    }
    else {
        this.model.destroy({
            success: function(data) {
                app.router.navigate("#", {trigger:true});
            },
            error: function(data, err, options) {
                var error = new Force.Error(err);
                alert("Failed to delete account:
                    " + (error.type === "RestError" ?
                         error.details[0].message :
                         "Remote change detected - delete aborted"));
            }
        });
    }
}
```

SyncPage

View for the sync page. This view monitors several events:

Event Type	Control	Handler function name
click	button-prev	goBack
click	sync	sync

To see how the screen is rendered, look at the render method:

```
render: function(eventName) {

    $(this.el).html(this.template(_.extend(
        {countLocallyModified: this.model.length},
        this.model.toJSON())));

    this.listView.setElement($("ul", this.el)).render();

    return this;

},
```

Let's take a look at what happens when the user taps **Process** (the sync control).

The `sync()` function looks at the first locally modified Account in the view's collection and tries to save it to the server. If the save succeeds and there are no more locally modified records, the app navigates back to the search screen. Otherwise, the app marks the account as having failed locally and then calls `sync()` again.

```
sync: function(event) {
    var that = this;
    if (this.model.length == 0 ||
        this.model.at(0).get("__sync_failed__")) {
        // We push sync failures back to the end of the list.
        // If we encounter one, it means we are done.
        return;
    }
    else {
        var record = this.model.shift();

        var options = {
            mergeMode: Force.MERGE_MODE.MERGE_FAIL_IF_CHANGED,
            success: function() {
                if (that.model.length == 0) {
                    app.router.navigate("#", {trigger:true});
                }
                else {
                    that.sync();
                }
            },
            error: function() {
                record = record.set("__sync_failed__", true);
                that.model.push(record);
                that.sync();
            }
        };
        return record.get("__locally_deleted__")
            ? record.destroy(options) :
            record.save(null, options);
    }
});
```

Router

When the router is initialized, it sets up the two global caches used throughout the sample.

```
setupCaches: function() {
    // Cache for offline support
    app.cache = new Force.StoreCache("accounts",
        [ {path:"Name", type:"string"} ]);

    // Cache for conflict detection
    app.cacheForOriginals = new Force.StoreCache("original-accounts");

    return $.when(app.cache.init(), app.cacheForOriginals.init());
},
```

Once the global caches are set up, it also sets up two `AccountCollection` objects: One for the search screen, and one for the sync screen.

```
// Collection behind search screen
app.searchResults = new app.models.AccountCollection();

// Collection behind sync screen
app.localAccounts = new app.models.AccountCollection();
app.localAccounts.config = {
    type:"cache",
    cacheQuery: {
        queryType:"exact",
        indexPath:"__local__",
        matchKey:true,
        order:"ascending",
        pageSize:25}};
```

Finally, it creates the view objects for the Search, Sync, and EditAccount screens.

```
// We keep a single instance of SearchPage / SyncPage and EditAccountPage
app.searchPage = new app.views.SearchPage({model: app.searchResults});
app.syncPage = new app.views.SyncPage({model: app.localAccounts});
app.editPage = new app.views.EditAccountPage();
```

The router has a `routes` field that maps actions to methods on the router class.

```
routes: {
    "": "list",
    "list": "list",
    "add": "addAccount",
    "edit/accounts/:id": "editAccount",
    "sync":"sync"
},
```

The `list` action fills the search result collections by calling `fetch()` and brings the search page into view.

```
list: function() {
    app.searchResults.fetch();
    // Show page right away - list will redraw when data comes in
    this.slidePage(app.searchPage);
},
```

The `addAccount` action creates an empty account object and bring the edit page for that account into view.

```
addAccount: function() {
    app.editPage.model = new app.models.Account({Id: null});
    this.slidePage(app.editPage);
},
```

The `editAccount` action fetches the specified Account object and brings the account detail page into view.

```
editAccount: function(id) {
    var that = this;
    var account = new app.models.Account({Id: id});
    account.fetch({
        success: function(data) {
            app.editPage.model = account;
```

```
            that.slidePage(app.editPage);
        },
        error: function() {
            alert("Failed to get record for edit");
        }
    });
}
```

The sync action computes the `localAccounts` collection by calling fetch and brings the sync page into view.

```
sync: function() {
    app.localAccounts.fetch();
    // Show page right away - list will redraw when data comes in
    this.slidePage(app.syncPage);
}
```

Validating Configuration Files

When you're writing formally structured text files, schema validation is useful at any level. For SmartStore and Mobile Sync configuration files, schema validation is especially welcome for complex configurations that handle related records.

Beginning in Mobile SDK 8.0, you can validate your configuration files using any of the following Mobile SDK utilities:

- forceios
- forcedroid
- forcereact
- Salesforce CLI Mobile SDK plugin

To validate your configurations, you use the `checkconfig` action.

Validating with Mobile SDK Utilities

Here's a Mobile SDK npm utility call to `checkconfig`.

```
$ forceios checkconfig
```

The tool then prompts you for the following information:

- `configpath`—Path to the configuration file
- `configtype`—Type of the configuration file. Must be either "store" or "syncs".

Validating with the Salesforce CLI Command Line

Here's an Salesforce CLI call to `checkconfig`.

```
$ sf mobilesdk:ios:checkconfig
```

You can replace `ios` with `android`, `hybrid`, or `reactnative`—in every case, though, the result is the same. You then provide the following required options:

- `-c, --configpath=`*<path to the configuration file>*
- `-y, --configtype=`*<either "store" or "syncs">*

Using Other Validators

If you prefer, you can use Mobile SDK schemas with third-party JSON validators. You can find the published schema definitions here:

- For store configurations: `store.schema.json`
- For sync configurations: `syncs.schema.json`

Schema Change History

Mobile SDK 9.1 Updates

Targets of type `soql` now accept an optional `maxBatchSize` property. This property accepts any integer between 200 and 2,000. Default value is 2,000.

Mobile SDK 8.0 Updates

To overcome differences between iOS and Android syncs configuration syntax, the iOS schema was revised as follows to match Android's syntax.

- The type for parent-children sync down configurations is now `parent_children`. Formerly, iOS used `parentChildren`.
- Create field list for parents in parent-children sync up configurations is labeled `createFieldlist`. Formerly, iOS used `parentCreateFieldlist`.
- Update field list for parents in parent-children sync up configurations is labeled `updateFieldlist`. Formerly, iOS used `parentUpdateFieldlist`.

Example: Sample configuration files:

- `userstore.json`
- `usersyncs.json`

CHAPTER 12 Files and Networking

In this chapter ...

Mobile SDK provides an API for files management that implements two levels of technology. For files management, Mobile SDK provides convenience methods that process file requests through the Connect REST API. Under the REST API level, networking classes give apps control over pending REST requests. Together, these two sides of the same coin give the SDK a robust content management feature as well as enhanced networking performance.

Architecture

Beginning with Mobile SDK 4.2, the Android REST request system uses OkHttp (v3.2.0), an open-source external library from Square Open Source, as its underlying architecture. This library replaces the Google Volley library from past releases. As a result, Mobile SDK no longer defines the `WrappedRestRequest` class. For more information, see square.github.io/okhttp/.

In iOS, file management and networking rely on the `SalesforceNetwork` library. All REST API calls—for files and any other REST requests—go through this library.

Note: If you directly accessed a third-party networking library in older versions of your app, update that code to use the `SalesforceNetwork` library.

Hybrid JavaScript functions use the the Mobile SDK architecture for the device operating system (Android, iOS, or Windows) to implement file operations. These functions are defined in `force.js`.

Downloading Files and Managing Sharing

Salesforce Mobile SDK provides convenience methods that build specialized REST requests for file download and sharing operations. You can use these requests to:

- Access the byte stream of a file.
- Download a page of a file.
- Preview a page of a file.
- Retrieve details of File records.
- Access file sharing information.
- Add and remove file shares.

Pages in Requests

The term "page" in REST requests can refer to either a specific item or a group of items in the result set, depending on the context. When you preview a page of a specific file, for example, the request retrieves the specified page from the rendered pages. For most other requests, a page refers to a section of the list of results. The maximum number of records or topics in a page defaults to 25.

The response includes a `NextPageUrl` field. If this value is defined, there is another page of results. If you want your app to scroll through pages of results, you can use this field to avoid sending unnecessary requests. You can also detect when you're at the end of the list by simply checking the response status. If nothing or an error is returned, there's nothing more to display and no need to issue another request.

Uploading Files

Native mobile platforms support a method for uploading a file. You provide a path to the local file to be uploaded, the name or title of the file, and a description. If you know the MIME type, you can specify that as well. The upload method returns a platform-specific request object that can upload the file to the server. When you send this request to the server, the server creates a file with version set to 1.

Use the following methods for the given app type:

App Type	Upload Method	Signature
Android native	`FileRequests.uploadFile()`	`public static RestRequest uploadFile( File theFile, String name, String description, String mimeType) throws UnsupportedEncodingException`
iOS native	`- requestForUploadFile: name:description:mimeType:`	`- (SFRestRequest *) requestForUploadFile:(NSData *)data name:(NSString *)name description:(NSString *)description mimeType:(NSString *)mimeType`
Hybrid (Android and iOS)	N/A	N/A

Encryption and Caching

Mobile SDK gives you access to the file's unencrypted byte stream but doesn't implement file caching or storage. You're free to devise your own solution if your app needs to store files on the device.

Using Files in Android Apps

The `FileRequests` class provides static methods for creating `RestRequest` objects that perform file operations. Each method returns the new `RestRequest` object. Applications then call the `ownedFilesList()` method to retrieve a `RestRequest` object. It passes this object as a parameter to a function that uses the `RestRequest` object to send requests to the server:

```
performRequest(FileRequests.ownedFilesList(null, null));
```

This example passes null to the first parameter (`userId`). This value tells the `ownedFilesList()` method to use the ID of the context, or logged-in, user. The second null, for the `pageNum` parameter, tells the method to fetch the first page of results.

For native Android apps, file management classes and methods live in the `com.salesforce.androidsdk.rest.files` package.

SEE ALSO:

Files API Reference

Managing the Request Queue

The `RestClient` class internally uses an instance of the `OkHttpClient` class to manage REST API requests. You can access underlying OkHttp objects directly to cancel pending requests. To manage a specific request, you can use the OkHttp `Call` object returned by the `RestClient.sendAsync()` Mobile SDK method.

Example: The following examples show how to perform some common network operations with `OkHttpClient`.

Common Imports

```
import okhttp3.Headers;
import okhttp3.HttpUrl;
import okhttp3.OkHttpClient;
import okhttp3.Call;
import okhttp3.Dispatcher;
import okhttp3.Request;
import okhttp3.RequestBody;
import okhttp3.Response;
```

Obtain the Current OkHttp Client Handle

To get the handle of the `OkHttpClient` that the current `RestClient` instance is using:

Kotlin

```
var okClient = restClient.getOkHttpClient()
```

Java

```
OkHttpClient okClient = restClient.getOkHttpClient();
```

Obtain the OkHttp Dispatcher

Kotlin

```
var dispatcher = restClient.getOkHttpClient().dispatcher()
```

Java

```
Dispatcher dispatcher = restClient.getOkHttpClient().dispatcher();
```

Cancel All Pending Calls

Kotlin

```
var dispatcher = restClient.getOkHttpClient().dispatcher()
dispatcher.cancelAll()
```

Java

```
Dispatcher dispatcher = restClient.getOkHttpClient().dispatcher();
                                dispatcher.cancelAll();
```

Store the OkHttp Handle to a REST Request

Kotlin

```
var call = restClient.sendAsync(restRequest, callback)
```

Java

```
Call call = restClient.sendAsync(restRequest, callback);
```

Cancel a Specific REST Request Using a Stored Handle

Kotlin

```
var call = restClient.sendAsync(restRequest, callback)
```

Java

```
Call call = restClient.sendAsync(restRequest, callback);
                                ...
                                call.cancel();
```

For more information, see square.github.io/okhttp/.

SEE ALSO:

OkHttp: The Underlying Network Library

Using Files in iOS Native Apps

To handle files in native iOS apps, use convenience methods defined in the `SFRestAPI (Files)` category. These methods parallel the files API for Android native and hybrid apps. They send requests to the same list of REST APIs, but use different underpinnings.

For example, the following code snippet calls the `requestForOwnedFilesList:page:` method to retrieve a `SFRestRequest` object. It then sends the request object to the server, specifying its owning object as the delegate that receives the response.

```
SFRestRequest *request = [[SFRestAPI sharedInstance] requestForOwnedFilesList:nil page:0];
[[SFRestAPI sharedInstance] send:request delegate:self];
...
```

This example passes nil to the first parameter (`userId`). This value tells the `requestForOwnedFilesList:page:` method to use the ID of the context, or logged in, user. Passing 0 to the `pageNum` parameter tells the method to fetch the first page.

Note: Swift versions of `SFRestAPI (Files)` methods are not defined explicitly by Mobile SDK. To code these methods in Swift, use the autocomplete suggestions offered by the Xcode compiler. These suggested method and parameter names are determined by Swift compiler heuristics and can differ from their Objective-C equivalents.

REST Responses and Multithreading

The SalesforceNetwork library always dispatches REST responses to the thread where your `SFRestDelegate` currently runs. This design accommodates your app no matter how your delegate intends to handle the server response. When you receive the response, you can do whatever you like with the returned data. For example, you can cache it, store it in a database, or immediately blast it to UI controls. If you send the response directly to the UI, however, remember that your delegate must dispatch its messages to the main thread.

SEE ALSO:

Files API Reference

Managing Requests

The `SalesforceNetwork` library for iOS defines two primary objects, `SFNetworkEngine` and `SFNetworkOperation`. `SFRestRequest` internally uses a `SFNetworkOperation` object to make each server call.

If you'd like to access the `SFNetworkOperation` object for any request, you have two options.

- The following methods return `SFNetworkOperation*`:
  - `[SFRestRequest send:]`
  - `[SFRestAPI send:delegate:]`
- `SFRestRequest` objects include a `networkOperation` object of type `SFNetworkOperation*`.

To cancel pending REST requests, you also have two options.

- `SFRestRequest` provides a new method that cancels the request:

```
- (void) cancel;
```

- And `SFRestAPI` has a method that cancels all requests currently running:

```
- (void)cancelAllRequests;
```

Example: **Examples of Canceling Requests**

To cancel all requests:

```
[[SFRestAPI sharedInstance] cancelAllRequests];
```

To cancel a single request:

```
SFRestRequest *request = [[SFRestAPI sharedInstance] requestForOwnedFilesList:nil
page:0];
[[SFRestAPI sharedInstance] send:request delegate:self];
...
// User taps Cancel Request button while waiting for the response
-(void) cancelRequest:(SFRestRequest *) request {
   [request cancel];
}
```

Using Files in Hybrid Apps

Hybrid file request wrappers reside in the `force+files.js` JavaScript library. When using the hybrid functions, you pass in a callback function that receives and handles the server response. You also pass in a function to handle errors.

To simplify the code, you can use the `mobilesync.js` and `force.js` libraries to build your HTML app. The FileExplorer sample app in the github.com/forcedotcom/SalesforceMobileSDK-Shared repo demonstrates this setup.

Note: Mobile SDK does not support file uploads in hybrid apps.

CHAPTER 13 Push Notifications and Mobile SDK

In this chapter ...

Notifications sent from Salesforce help your mobile users stay on top of important developments in their organizations. Salesforce notification services let you configure and test mobile push notifications before you implement any code. To receive mobile notifications in a production environment, your Mobile SDK app registers with the mobile OS provider and then registers for Salesforce notifications. Mobile SDK minimizes your coding effort by implementing most of the registration tasks internally.

Mobile SDK supports Salesforce Notification Builder custom notifications at no cost to your client app. Mobile SDK apps don't require extra coding to receive and display Notification Builder push notifications. However, if you choose to implement a message tray in your app, Mobile SDK provides convenience methods for calling the required Salesforce APIs.

Notification Builder notifications arrive encrypted. Mobile SDK decrypts messages internally, requiring only the addition of a boilerplate class extension in iOS apps and no additional coding on Android. Apex push notifications are not encrypted and remain fully supported.

About Push Notifications

With the Salesforce notification service, you can develop and test push notifications in Mobile SDK custom apps. Mobile SDK provides APIs that you can implement to register devices with the push notification service. However, receiving and handling the notifications remain the responsibility of the developer.

Setup to receive push notification occurs on several levels:

- Configuring push services from the device technology provider (Apple for iOS, Google for Android)
- Configuring your Salesforce connected app definition to enable push notifications
- Configuring the Salesforce org to send push notifications to your app through one or more of these methods:
  - Implementing Apex triggers
  - Subscribing to Notification Builder custom notification types and pushing those notifications through Process Builder, Flow Builder, or the Connect REST API
  - Calling the push notification resource of the Connect REST API
- Making minor code changes to support registration in your Mobile SDK app
- Implementing a class extension to support decryption in iOS apps
- Registering the mobile device at runtime

You're responsible for Apple or Google service configuration, connected app configuration, Apex or Connect REST API coding, and minor changes to your Mobile SDK app. Salesforce Mobile SDK handles runtime registration transparently.

For information on setting up mobile push notifications for your organization and creating your own custom notifications tray, see the Mobile Notifications Implementation Guide.

Using Push Notifications in Hybrid Apps

To use push notifications in a hybrid app, first be sure to

- Register for push notifications with the OS provider.
- Configure your connected app to support push notifications for your target device platform.

Salesforce Mobile SDK lets your hybrid app register itself to receive notifications, and then you define the behavior that handles incoming notifications.

SEE ALSO:

Using Push Notifications in Android

Using Push Notifications in iOS

Code Modifications (Hybrid)

1. (Android only) If your target platform is Android:
   a. Add an entry for `androidPushNotificationClientId`. In `assets/www/bootconfig.json`:

   ```
   "androidPushNotificationClientId": "33333344444"
   ```

   This value is the project number of the Google project that is authorized to send push notifications to an Android device.

2. In your callback for `cordova.require("com.salesforce.plugin.oauth").getAuthCredentials()`, add the following code:

```
cordova.require("com.salesforce.util.push").registerPushNotificationHandler(
    function(message) {
        // add code to handle notifications
    },
    function(error) {
        // add code to handle errors
    }
);
```

Example: This code demonstrates how you might handle messages. The server delivers the payload in `message["payload"]`.

```
function(message) {
    var payload = message["payload"];
    if (message["foreground"]) {
        // Notification is received while the app is in
        // the foreground
        // Do something appropriate with payload
    }
    if (!message["foreground"]) {
        // Notification was received while the app was in
        // the background, and the notification was clicked,
        // bringing the app to the foreground
        // Do something appropriate with payload
    }
}
```

Using Push Notifications in Android

Salesforce sends push notifications to Android apps through the Firebase Cloud Messaging (FCM) framework. See Firebase Cloud Messaging for an overview of this framework.

When developing an Android app that supports push notifications, remember these key points:

- You must
  - be signed in to a Google account.
  - have access to Firebase.
- To test FCM push services, we recommend using an Android physical device with either the Android Market app or Google Play Services installed. Push notifications are less reliable on emulators and work only on the "Android with Google Play Services" emulator type.
- You can also use the Send Test Notification link in your connected app detail view to perform a "dry run" test without pinging a device. You can also use this feature with Notification Builder push notifications.

To begin, create a Google API project for your app. Your project must have the FCM for Android feature enabled. See Firebase Cloud Messaging for instructions on setting up your project.

The setup process for your Google API project creates a key for your app. Once you've finished the project configuration, add the FCM key to your connected app settings.

Note: Push notification registration occurs at the end of the OAuth login flow. Therefore, an app does not receive push notifications unless and until the user logs into a Salesforce organization.

Configure a Connected App For FCM (Android)

To configure your Salesforce connected app to support push notifications:

1. If you have an existing Firebase project associated with your app, open the existing project. If not, create a project for your app in the Google Firebase Console.

   Important: If you previously set up notifications for your Android app, make sure that you proceed with the next steps using the existing Firebase project associated with your app. To avoid any disruption in your app's notifications, confirm that the sender ID in the Firebase project matches your app's existing sender ID.

2. In the Firebase project, click the cog icon next to Project Overview, and then click **Project settings**.
3. Collect the Firebase project ID.
   a. Click the **General** tab.
   b. Record the value in the Project ID field. You need the project ID for a later step.
4. Generate an Admin SDK private key for your Firebase service account.
   a. Select the **Service accounts** tab, and then click **Generate new private key**.
   b. Download the JSON file that contains the private key. Note the location of the downloaded file, because you need it later.

      The file name format for this private key is similar to `Pc-api-1234567890123456789-123-firebase-adminsdk-a1bcd-a1234bc5678.json`.
5. Create your mobile connected app in App Manager.
6. In the Mobile App Settings section of your mobile connected app, complete these fields.
   a. In the **App Platform** field, select **Android**.
   b. Select **Push Messaging**.
   c. In the **Platform** field, select **Android**.
   d. In the **Firebase Admin SDK Private Key** field, upload the JSON file that contains the private key that you generated for your Firebase service account.
   e. In the **Project ID** field, enter the project ID that you collected from the Firebase project.

   Note: As of Spring '24, the Mobile App Settings section includes a field for the legacy Firebase Cloud Messaging API server key. However, legacy Cloud Messaging API users are required to migrate to the new Firebase Cloud Messaging API (HTTP v1) by June 2024. We recommend that you submit the Firebase Admin SDK private key and project ID required for the new Firebase Cloud Messaging API.
7. Save your changes.

Code Modifications (Android)

To configure your Mobile SDK app to support push notifications:

1. Add an entry for `androidPushNotificationClientId`.

- In `res/values/bootconfig.xml` (for native apps):

  ```
  <string name="androidPushNotificationClientId">33333344444</string>
  ```

- In `assets/www/bootconfig.json` (for hybrid apps):

  ```
  "androidPushNotificationClientId": "33333344444"
  ```

  This value is the project number of the Google project that is authorized to send push notifications to an Android device.

  Behind the scenes, Mobile SDK automatically reads this value and uses it to register the device against the Salesforce connected app. This validation allows Salesforce to send notifications to the connected app. At logout, Mobile SDK also automatically unregisters the device for push notifications.

2. Create a class in your app that implements `PushNotificationInterface`. `PushNotificationInterface` is a Mobile SDK Android interface for handling push notifications. `PushNotificationInterface` has a single method, `onPushMessageReceived(Bundle message)`:

   ```
   public interface PushNotificationInterface {
       public void onPushMessageReceived(Bundle message);
   }
   ```

   In this method you implement your custom functionality for displaying, or otherwise disposing of, push notifications.

3. In the `onCreate()` method of your `Application` subclass, call the `SalesforceSDKManager.setPushNotificationReceiver()` method, passing in your implementation of `PushNotificationInterface`. Call this method immediately after the `SalesforceSDKManager.initNative()` call. For example:

   ```
   @Override
   public void onCreate() {
      super.onCreate();
      SalesforceSDKManager.initNative(getApplicationContext(),
         new KeyImpl(), MainActivity.class);
      SalesforceSDKManager.getInstance().
         setPushNotificationReceiver(myPushNotificationInterface);
   }
   ```

In Android apps, decryption of Notification Builder push notifications occurs automatically. Apex push notifications are not encrypted.

Using Push Notifications in iOS

When developing an iOS app that supports push notifications, remember these key points:

- You must be a member of the iOS Developer Program.
- You can test Apple push services only on an iOS physical device. Push notifications don't work in the iOS simulator.
- There are no guarantees that all push notifications will reach the target device, even if the notification is accepted by Apple.
- Apple Push Notification Services setup requires the use of the OpenSSL command line utility provided in Mac OS X.

Before you can complete registration on the Salesforce side, you need to register with Apple Push Notification Services. The following instructions provide a general outline for what's required. See http://www.raywenderlich.com/32960/ for complete instructions.

Configuration for Apple Push Notification Services

Registering with Apple Push Notification Services (APNS) requires the following items.

Certificate Signing Request (CSR) File
Generate this request using the Keychain Access feature in Mac OS X. You'll also use OpenSSL to export the CSR private key to a file for later use.

App ID from iOS Developer Program
In the iOS Developer Member Center, create an ID for your app, then use the CSR file to generate a certificate. Next, use OpenSSL to combine this certificate with the private key file to create a `.p12` file. You'll need this file later to configure your connected app.

iOS Provisioning Profile
From the iOS Developer Member Center, create a new provisioning profile using your iOS app ID and developer certificate. You then select the devices to include in the profile and download to create the provisioning profile. You can then add the profile to Xcode. Install the profile on your test device using Xcode's Organizer.

When you've completed the configuration, sign and build your app in Xcode. Check the build logs to verify that the app is using the correct provisioning profile. To view the content of your provisioning profile, run the following command at the Terminal window:
`security cms -D -i <your profile>.mobileprovision`

Configure a Connected App for APNS (iOS)

For iOS versions of your mobile app, you can use tokens, push certificates, or both. If a token is provided, Salesforce always uses the token.

1. Create your mobile connected app.
2. For App Platform, select **iOS**.
3. Select **Push Messaging Enabled**.
4. For Platform, select **Apple**.
5. Optionally provide an Application Bundle ID to avoid problems with your push notification settings.
6. Select the environment to enable push notifications for.
7. Upload the Signing Key from your Apple developer account.
8. Enter the Key Identifier from your Apple developer account.
9. Enter the Team Identifier from your Apple developer account.

Code Modifications (iOS)

To handle notifications in iOS apps, you register in the `AppDelegate` class using the provided template code. In Swift apps, you must add an extension to decrypt incoming Notification Builder notifications.

Registering to Receive Notifications

Mobile SDK for iOS provides the `SFPushNotificationManager` class to handle push registration. To use it in Objective-C, import `<SalesforceSDKCore/SFPushNotificationManager>`. Swift doesn't require a special import.

The `SFPushNotificationManager` class is available as a runtime shared instance:

Swift

```
SFPushNotificationManager.sharedInstance()
```

Objective-C

```
[SFPushNotificationManager sharedInstance]
```

This class implements these registration methods:

Swift

```
func registerForRemoteNotifications()

func application(_ application: UIApplication,
    didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data)

func registerSalesforceNotifications(completionBlock: (() → Void)?,
    fail: (() → Void)?)

func unregisterSalesforceNotifications(withCompletionBlock: UserAccount,
    completionBlock: (() → Void)?) // for internal use
```

Objective-C

```
- (void)registerForRemoteNotifications;

- (void)didRegisterForRemoteNotificationsWithDeviceToken:
    (NSData*)deviceTokenData;

- (BOOL)registerSalesforceNotificationsWithCompletionBlock:(nullable
    void (^)(void))completionBlock failBlock:(nullable void (^)(void))failBlock;

- (BOOL)unregisterSalesforceNotificationsWithCompletionBlock:(SFUserAccount*)user
    completionBlock:(nullable void (^)(void))completionBlock; // for internal use
```

Mobile SDK calls `unregisterSalesforceNotifications` at logout.

Note:

- Salesforce encrypts Notification Builder notifications.
- To support full content push notifications, your iOS app must implement the decryption class extension.
- If a Mobile SDK app that hasn't implemented decryption receives an encrypted Salesforce notification, the customer sees only the notification title.
- Salesforce does not encrypt Apex push notifications.

Implementing a Decryption Extension (Swift)

In Mobile SDK 8.2 and later, the forceios iOSNativeSwiftTemplate app requests notification authorization through the iOS `UNUserNotificationCenter` object. A specialized version of `iOSNativeSwiftTemplate`, iOSNativeSwiftEncryptedNotificationTemplate, extends the `UNNotificationServiceExtension` iOS class to handle notification decryption. This extension class, `NotificationService`, provides boilerplate decryption code that Mobile SDK apps can use without changes. To support encrypted notifications, you must be using Mobile SDK 8.2 or later, and your app must include this extension.

To create a Swift project that supports Notification Builder encrypted notifications, you can use the `iOSNativeSwiftEncryptedNotificationTemplate` template with forceios. Even if you're updating an existing Mobile SDK project, it's easiest to start fresh with a new forceios template project. If you'd rather update a Swift project manually, skip to "Example: Add Push Registration Manually (Swift)".

1. Install the latest forceios version from node.js:

```
[sudo] npm install -g forceios
```

2. Call the forceios `createWithTemplate` command:

```
forceios createWithTemplate
```

3. At the first prompt, enter *`iOSNativeSwiftEncryptedNotificationTemplate`*:

```
forceios createWithTemplate
Enter URI of repo containing template application or a Mobile SDK template name:
    iOSNativeSwiftEncryptedNotificationTemplate
```

4. In the remaining prompts, enter your company and project information. If your information is accepted, forceios creates a project that is ready for encrypted notifications.
5. If you're updating an older Mobile SDK project, copy your app-specific resources from your old project into the new project.

Example: **Add Push Registration Manually (Swift)**

This example requires an existing app built on Mobile SDK 8.2 or higher and based on the `iOSSwiftNativeTemplate` project.

Configure Push Notification Registration in `AppDelegate`

1. In the `application(_:didFinishLaunchingWithOptions:)` method, uncomment the call to `registerForRemotePushNotifications`.

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
    launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

    self.window = UIWindow(frame: UIScreen.main.bounds)
    self.initializeAppViewState()
    // If you wish to register for push notifications, uncomment the line below.
    // Note that if you want to receive push notifications from Salesforce,
    // you will also need to implement the
    // application(application, didRegisterForRemoteNotificationsWithDeviceToken)
    // method (below).
    //
    self.registerForRemotePushNotifications()
    ...
```

   The `registerForRemotePushNotifications` method attempts to register your app with Apple for receiving remote notifications. If registration succeeds, Apple passes a device token to the `application(_:didRegisterForRemoteNotificationsWithDeviceToken:)` method of your `AppDelegate` class.

2. In the `application(_:didRegisterForRemoteNotificationsWithDeviceToken:)` method, uncomment the call to `didRegisterForRemoteNotifications(withDeviceToken:)`.

```
func application(_ application: UIApplication,
    didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data) {
    // Uncomment the code below to register your device token with the
    // push notification manager
    //
        didRegisterForRemoteNotifications(deviceToken)
```

```
        ...
    }
    ...
```

3. To log a debugger error if registration with Apple fails, add the following code to the `application(_:didFailToRegisterForRemoteNotificationsWithError:)` method.

```
func application(_ application: UIApplication,
    didFailToRegisterForRemoteNotificationsWithError error: Error ) {

    // Respond to any push notification registration errors here.
    SalesforceLogger.d(type(of:AppDelegate.self),
        message:"Failed to get token, error: \(error)")
    }
}
```

Add the Decryption Extension

1. In the Project navigator, select the target where you modified `AppDelegate`.
2. In Project settings, select **Signing & Capabilities**.
3. Click **+ Capability** and search for "Push Notifications". Double-click to add the capability.
4. Follow the steps at "Add a Service App Extension to Your Project" in *Modifying Content in Newly Delivered Notifications* at `developer.apple.com/documentation`.
5. If you plan to test the extension with its own settings, you can accept the prompt to activate a scheme. Otherwise, click **Cancel**.
6. Clone or download the github.com/forcedotcom/SalesforceMobileSDK-Templates repo.
7. From the `iOSNativeSwiftEncryptedNotificationTemplate/NotificationServiceExtension` folder, replace the code in your new extension's Swift file with the code from `NotificationServiceExtension.swift`.

CHAPTER 14 Authentication, Security, and Identity in Mobile Apps

In this chapter ...

Secure authentication is essential for enterprise applications running on mobile devices. OAuth 2.0, the industry-standard protocol, enables secure authorization for access to a customer's data, without handing out the username and password. It is often described as the valet key of software access. A valet key restricts access to certain features of your car. For example, a parking attendant can't open the trunk or glove compartment using a valet key.

Mobile app developers can quickly and easily embed the Salesforce OAuth 2.0 implementation. The implementation uses an HTML view to collect the username and password, which are then sent to the server. The server returns a session token and a persistent refresh token that are stored on the device for future interactions.

A Salesforce *connected app* is the primary means by which a mobile app connects to Salesforce. A connected app gives both the developer and the administrator control over how the app connects and who has access. For example, a connected app can restrict access to a set of customers, set or relax an IP range, and so on.

OAuth Terminology

Access Token
A value used by the consumer to gain access to protected resources on behalf of the user, instead of using the user's Salesforce credentials. The access token is a session ID, and can be used directly.

Authorization Code
A short-lived token that represents the access granted by the end user. The authorization code is used to obtain an access token and a refresh token.

Connected App
An application external to Salesforce that uses the OAuth protocol to verify both the Salesforce user and the external application.

Consumer Key
A value used by the consumer—in this case, the Mobile SDK app—to identify itself to Salesforce. Referred to as `client_id`.

Consumer Secret
A secret that the consumer uses to verify ownership of the consumer key. To heighten security, Mobile SDK apps do not use the consumer secret.

Refresh Token
A token used by the consumer to obtain a new access token, without having the end user approve the access again.

Remote Access Application (DEPRECATED)
A *remote access application* is an application external to Salesforce that uses the OAuth protocol to verify both the Salesforce user and the external application. A remote access application is implemented as a connected app. Remote access applications have been deprecated in favor of connected apps.

OAuth 2.0 Authentication Flow

The authentication flow depends on the state of authentication on the device. The following steps assume that Salesforce authentication occurs at app startup.

First Time Authorization Flow

1. The customer opens a Mobile SDK app.
2. An authentication prompt appears.
3. The customer enters a username and password.
4. The app sends the customer's credentials to Salesforce and, in return, receives a session ID as confirmation of successful authentication.
5. The customer approves the app's request to grant access to the app.
6. The app starts.

Ongoing Authorization

1. The customer opens a mobile app.
2. If the session ID is active, the app starts immediately. If the session ID is stale, the app uses the refresh token from its initial authorization to get an updated session ID.
3. The app starts.

PIN Authentication (Optional)

PIN protection is a function of the mobile policy and is used only when it's enabled in the Salesforce connected app definition. See About PIN Security.

OAuth 2.0 Web Server Flow

Beginning in Mobile SDK 11.0, OAuth 2.0 Web Server Flow is the default authentication flow. Upon upgrading to Mobile SDK 11.0, you do not need to make any changes in your client application. However, make sure the "Require Secret for Web Server Flow" checkbox is deselected in your connected app.

Prior to Mobile SDK 11.0, an intermediary authorization screen prompts the user to approve or deny the authorization request on each login attempt. With the upgrade to Web Server Flow in 11.0, this screen is shown only the first time a user authorizes the connected app (assuming the app is configured to allow the user to self-authorize). This behavior change is unlikely to impact day-to-day use, but it could impact areas such as automated tests, should they be written to anticipate this intermediary screen in the login flow.

For more information on using Web Server Flow, visit OAuth 2.0 Web Server Flow for Web App Integration.

Opting Out for User-Agent Flow

To opt out of your Web Server Flow in Mobile SDK 11.0 and on, you can revert to User-Agent Flow on page 405 in the `SalesforceSDKManager`.

Android

```
SalesforceSDKManager.getInstance().setUseWebServerAuthentication(false)
```

iOS

Swift

```
SalesforceManager.shared.useWebServerAuthentication = false
```

Objective-C

```
[SalesforceSDKManager sharedManager].useWebServerAuthentication = NO;
```

OAuth 2.0 User-Agent Flow

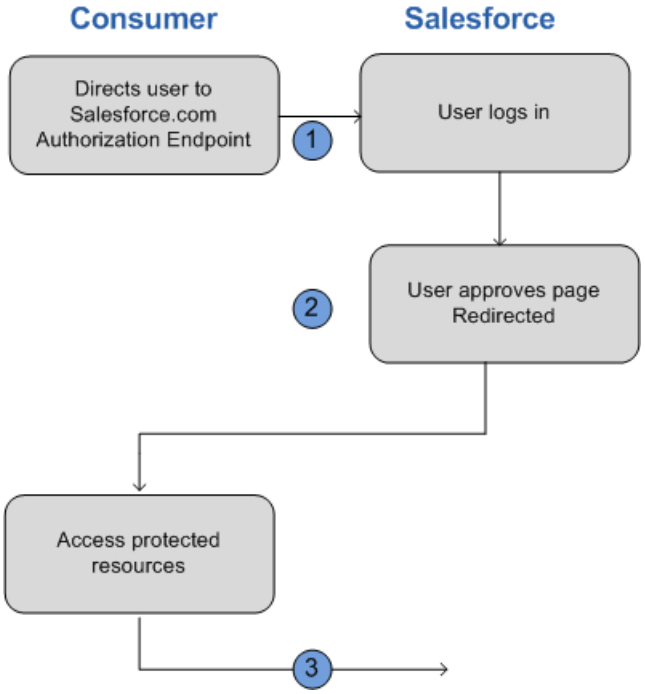
With the OAuth 2.0 user-agent flow, users authorize a desktop or mobile app to access data by using an external or embedded browser. Client apps running in a browser using a scripting language such as JavaScript can also use this flow. This flow uses the OAuth 2.0 implicit grant type.

In the user-agent flow, the connected app, which integrates the client app with the Salesforce API, receives the access token as an HTTP redirection. The connected app requests that the authorization server redirects the user-agent to a web server or to an accessible local resource. The web server can extract the access token from the response and pass it to the connected app. For security, the token response is provided as a hash tag (#) fragment on the URL. This format prevents the token from being passed to the server or to any other servers in referral headers.

Warning: Because the access token is encoded into the redirection URL, it can be exposed to the user and other apps on the device.

If you're using JavaScript to authenticate, call `window.location.replace();` to remove the callback from the browser's history.

Note: Connected apps for these types of clients can protect per-user secrets. But the client secret is accessible and exploitable because client executables reside on the user's device. For this reason, the user-agent flow doesn't use the client secret. Authorization is based on the user-agent's same-origin policy. Also, the user-agent flow doesn't support out-of-band posts.



For example, you use Salesforce Mobile SDK to build a mobile app that looks up customer contact information from your Salesforce org. Mobile SDK implements the OAuth 2.0 user-agent flow for your connected app, integrating the mobile app with your Salesforce API and giving it authorized access to the defined data. The flow follows these steps.

1. The user opens the mobile app.
2. The connected app directs the user to Salesforce to authenticate and authorize the mobile app.
3. The user approves access for this authorization flow.
4. The connected app receives the callback from Salesforce to the redirect URL, which extracts the access and refresh tokens.
5. The connected app uses the access token to access data on the user's behalf.

User-Agent Flow in Mobile SDK 11.0 and On

Note: Up until Mobile SDK 10.2, OAuth 2.0 User-Agent Flow is used for web view authentication on both iOS and Android and for advanced authentication on Android. Starting in Mobile SDK 11.0, the default authentication on both platforms uses the OAuth 2.0 Web Server Flow on page 405 with Proof Key for Code Exchange (PKCE) for increased security.

If you want to continue using User-Agent Flow for your app in Mobile SDK 11.0 and on, the option can be enabled in the `SalesforceSDKManager`. Before initiating a login, call one of the following methods in your application class's `init` method.

Android

```
SalesforceSDKManager.getInstance().setUseWebServerAuthentication(false)
```

iOS

Swift

```
SalesforceManager.shared.useWebServerAuthentication = false
```

Objective-C

```
[SalesforceSDKManager sharedManager].useWebServerAuthentication = NO;
```

OAuth 2.0 Refresh Token Flow

After a client—via a connected app—receives an access token, it can use a refresh token to get a new session when its current session expires. The connected app's session timeout value determines when an access token is no longer valid and when to apply for a new one using a refresh token.

The refresh token flow involves these steps.

Note: Mobile SDK apps can use the SmartStore feature to store data locally for offline use. SmartStore data is inherently volatile. Its lifespan is tied to the authenticated user as well as to OAuth token states. When the user logs out of the app, SmartStore deletes all soup data associated with that user. Similarly, when the OAuth refresh token is revoked or expires, the user's app state is reset, and all data in SmartStore is purged. Carefully consider the volatility of SmartStore data when designing your app. This warning is especially important if your org sets a short lifetime for the refresh token.

Mobile SDK Login and Authentication Flow—Detailed Look

Mobile SDK handles the complex login and authentication flow internally so that you don't have to orchestrate it yourself. If your app is reporting login failures, use this detailed description to help pinpoint the problem.

Scope Parameter Values

OAuth requires scope configuration both on server and on client. The agreement between the two sides defines the scope contract.

- **Server side**—Define scope permissions in a connected app on the Salesforce server. These settings determine which levels of access client apps, such as Mobile SDK apps, can request. At a minimum, configure your connected app OAuth settings to match what's specified in your code. For most apps, `refresh_token`, `web`, and `api` are sufficient.
- **Client side**—Specify scope requests in your Mobile SDK app. Client scope requests must be a subset of the connected app's scope permissions.

Server Side Configuration

You can set the following scope parameter values.

Value	Description
Perform ANSI SQL queries on Customer Data Platform data (`cdp_query_api`)	Allows ANSI SQL queries of Data Cloud data on behalf of the user.
Manage Pardot services (`pardot_api`)	Allows access to Marketing Cloud Account Engagement API services on behalf of the user. Manage the full extent of accessible services

Value	Description
	in Account Engagement. (Pardot is now Marketing Cloud Account Engagement.)
Manage Customer Data Platform profile data (`cdp_profile_api`)	Allows access to Data Cloud REST API data. Use this scope to manage profile records.
Access Connect REST API resources (`chatter_api`)	Allows access to Connect REST API resources on behalf of the user.
Manage Customer Data Platform Ingestion API data (`cdp_ingest_api`)	Allows access to Data Cloud Ingestion API data. Use this scope to upload and maintain external datasets in Data Cloud. This scope is packaged in a JSON web token (JWT).
Access Analytics REST API Charts Geodata resources (`eclair_api`)	Allows access to the Analytics REST API Charts Geodata resource.
Access Analytics REST API resources (`wave_api`)	Allows access to the Analytics REST API resources.
Manage user data via APIs (`api`)	Allows access to the current, logged-in user's account using APIs, such as REST API and Bulk API 2.0. This scope also includes `chatter_api`, which allows access to Connect REST API resources.
Access custom permissions (`custom_permissions`)	Allows access to the custom permissions in an org associated with the connected app. This scope also shows whether the current user has each permission enabled.
Access the identity URL service (`id, profile, email, address, phone`)	Allows access to the identity URL service. You can request `profile`, `email`, `address`, or `phone` individually to get the same result as using `id` because they're synonymous.
Access Lightning applications (`lightning`)	Allows hybrid apps to directly obtain Lightning child sessions through the OAuth 2.0 hybrid app token flow and hybrid app refresh token flow.
Access content resources (`content`)	Allows hybrid apps to directly obtain content child sessions through the OAuth 2.0 hybrid app token flow and hybrid app refresh token flow.
Access unique user identifiers (`openid`)	Allows access to the current, logged in user's unique identifier for OpenID Connect apps. In the OAuth 2.0 user-agent flow and the OAuth 2.0 web server flow, use the `openid` scope. In addition to the access token, this scope enables you to receive a signed ID token that conforms to the OpenID Connect specifications.
Full access (`full`)	Allows access to all data accessible by the logged-in user, and encompasses all other scopes. `full` doesn't return a refresh token. You must explicitly request the `refresh_token` scope to get a refresh token.

Value	Description
Perform requests at any time (`refresh_token, offline_access`)	Allows a refresh token to be returned when the requesting client is eligible to receive one. With a refresh token, the app can interact with the user's data while the user is offline. This token is synonymous with requesting `offline_access`.
Access Visualforce applications (`visualforce`)	Allows access to customer-created Visualforce pages only. This scope doesn't allow access to standard Salesforce UIs. To allow hybrid apps to directly obtain Visualforce child sessions, include this scope with the OAuth 2.0 hybrid app token flow or hybrid app refresh token flow.
Manage user data via Web browsers (`web`)	Allows use of the `access_token` on the web. This scope also includes `visualforce`, allowing access to customer-created Visualforce pages.
Access chatbot services (`chatbot_api`)	Allows access to Einstein Bot API services.
Access Headless Registration API (user_registration_api)	Allows access to the API for the Headless Registration Flow. If you set up your flow to require authentication, you must pass in an access token that includes this scope.
Access Headless Forgot Password API (`forgot_password`)	Allows access to the API for the Headless Forgot Password Flow. If you set up your flow to require authentication, you must pass in an access token that includes this scope.
Access all Data Cloud API resources (`cdp_api`)	Allows access to all Data Cloud API resources.
Access the Salesforce API Platform (`sfap_api`)	Reserved for future use.
Access Interaction API resources (interaction_api)	Reserved for future use.

Note: For Mobile SDK apps, you're always required to select `refresh_token` in server-side Connected App settings. Even if you select the `full` scope, you still must explicitly select `refresh_token`.

Client Side Configuration

The following rules govern scope configuration for Mobile SDK apps.

Scope	Mobile SDK App Configuration
`refresh_token`	Implicitly requested by Mobile SDK for your app; no need to include in your app's list of scopes.
`api`	Include if you're making any Salesforce REST API calls (applies to most apps).

Scope	Mobile SDK App Configuration
`web`	Include if your app accesses pages defined in a Salesforce org (for any app that loads Salesforce-based web pages.)
`full`	Include to request all permissions. (Mobile SDK implicitly requests `refresh_token` for you.)
`chatter_api`	Include if your app calls Connect REST APIs.
`id`	(Not needed)
`visualforce`	Use `web` instead.

Using Identity URLs

The Identity URL is returned in the `id` scope parameter. For example, `https://login.salesforce.com/id/00Dx0000000BV7z/005x00000012Q9P`.

The identity URL is also a RESTful API to query for additional information about users, such as their username, email address, and org ID. It also returns endpoints that the client can talk to, such as photos for profiles and accessible API endpoints.

The format of the URL is `https://login.salesforce.com/id/`*`orgID`*`/`*`userID`*, where *`orgId`* is the ID of the Salesforce org that the user belongs to and *`userID`* is the Salesforce user ID.

Identity URL Request Parameters

You can use the following parameters with the access token and identity URL. You can use the access token in an authorization request header or a request with the `oauth_token` parameter.

Parameter	Description
`access_token`	OAuth token that a connected app uses to request access to a protected resource on behalf of the client application. Additional permissions in the form of scopes can accompany the access token.
`format`	Optional. Specify the format of the returned output. Values are: • `json` • `xml` The client can also specify the returned format in an accept-request header using one of the following formats. • `Accept: application/json` • `Accept: application/xml` • `Accept: application/x-www-form-urlencoded` The request header also supports the following.

Parameter	Description
	• The `*/*` wildcard is accepted and returns JSON. • A list of values, which is checked left to right. For example: `application/xml,application/json,application/html,*/*` returns XML. The `format` parameter takes precedence over the access request header.
`version`	Optional. Specify a SOAP API version number or the literal string `latest`. If this value isn't specified, the returned API URLs contain the literal value `{version}` in place of the version number. If the value is specified as `latest`, the most recent API version is used.
`PrettyPrint`	Optional. Accepted only in a header and not as a URL parameter. Specify this parameter to optimize the returned XML or JSON output for readability rather than size. For example, use the following in a header: `X-PrettyPrint:1`.
`callback`	Optional. Specify a valid JavaScript function name. You can use this parameter when the specified format is JSON. The output is wrapped in this function name (JSONP). For example, if a request to `https://server/id/orgid/userid/` returns `{"function":"name"}`, a request to `https://server/id/orgid/userid/?callback=baz` returns `baz({"function":"name"});`. Note: JSONP is no longer returned for Identity Service requests due to strict MIME typing. Your requests must add `'format=jsonp'` with the callback parameter so that the Identity Service returns JavaScript. When the Identity Service detects the JSONP format, it returns the required JavaScript type (`'application/javascript'`).

Identity URL Response

With a successful request, the identity URL response returns information about the queried user and org.

The following identity URL response is in XML format.

```
<?xml version="1.0" encoding="UTF-8"?>
<user xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<id>https://MyDomainName.my.salesforce.com/id/00Dx000.../005x000...</id>
<asserted_user>true</asserted_user>
<user_id>005x000000...</user_id>
<organization_id>00Dx000...</organization_id>
<nick_name>admin1.2777578168398293E12...</nick_name>
<display_name>Alan Van</display_name>
<email>admin@mycompany.com</email>
<status>
   <created_date xsi:nil="true"/>
   <body xsi:nil="true"/>
</status>
<photos>
   <picture>https://MyDomainName--03925205UAF.file.force-user-content.com</picture>
   <thumbnail>https://MyDomainName--03925205UAF.file.force-user-content.com</thumbnail>
</photos>
```

```
<urls>

<enterprise>https://MyDomainName.my.salesforce.com/services/Soap/c/{version}/00Dx000...</enterprise>


<metadata>https://MyDomainName.my.salesforce.com/services/Soap/m/{version}/00Dx000...</metadata>


<partner>https://MyDomainName.my.salesforce.com/services/Soap/u/{version}/00Dx000...</partner>

   <rest>https://MyDomainName.my.salesforce.com/services/data/v{version}/
   </rest>
   <sobjects>https://MyDomainName.my.salesforce.com/services/data/v{version}/sobjects/
   </sobjects>
   <search>https://MyDomainName.my.salesforce.com/services/data/v{version}/search/
   </search>
   <query>https://MyDomainName.my.salesforce.com/services/data/v{version}/query/
   </query>
   <profile>https://MyDomainName.my.salesforce.com/005x000...</profile>
</urls>
<active>true</active>
<user_type>STANDARD</user_type>
<language>en_US</language>
<locale>en_US</locale>
<utcOffset>-28800000</utcOffset>
<last_modified_date>2021-04-28T20:54:09.000Z</last_modified_date>
</user>
```

And this response is in JSON format.

```
{"id":"https://MyDomainName.my.salesforce.com/id/00Dx000.../005x000...",
"asserted_user":true,
"user_id":"005x000...",
"organization_id":"00Dx000...",
"nick_name":"admin1.2777578168398293E12...",
"display_name":"Alan Van",
"email":"admin@mycompany.com",
"status":{"created_date":null,"body":null},
"photos":{"picture":"https://MyDomainName--03925205UAF.file.force-user-content.com",
   "thumbnail":"https://MyDomainName--03925205UAF.file.force-user-content.com"},
"urls":

{"enterprise":"https://MyDomainName.my.salesforce.com/services/Soap/c/{version}/00Dx000...",

  "metadata":"https://MyDomainName.my.salesforce.com/services/Soap/m/{version}/00Dx000...",

  "partner":"https://MyDomainName.my.salesforce.com/services/Soap/u/{version}/00Dx000...",

   "rest":"https://MyDomainName.my.salesforce.com/services/data/v{version}/",
  "sobjects":"https://MyDomainName.my.salesforce.com/services/data/v{version}/sobjects/",

   "search":"https://MyDomainName.my.salesforce.com/services/data/v{version}/search/",
   "query":"https://MyDomainName.my.salesforce.com/services/data/v{version}/query/",
   "profile":"https://MyDomainName.my.salesforce.com/005x000..."},
```

```
"active":true,
"user_type":"STANDARD",
"language":"en_US",
"locale":"en_US",
"utcOffset":-28800000,
"last_modified_date":"2021-04-28T20:54:09.000+0000"}
```

This table describes the returned parameters.

Parameter	Description
`id`	Identity URL, which is the same URL that was queried.
`asserted_user`	Boolean value indicating whether the specified access token was issued for this identity.
`user_id`	User ID of the queried user.
`username`	Username of the queried user.
`organization_id`	ID of the queried user's Salesforce org.
`nick_name`	Experience Cloud nickname of the queried user.
`display_name`	Display name (full name) of the queried user.
`email`	Email address of the queried user.
`email_verified`	Indicates whether the queried user's email was verified by clicking a link in the "Welcome to Salesforce" email. The `email_verified` value is set to `true` when users click a link in the email they receive after the following:For example, a Salesforce admin creates the user Roberta Smith. Roberta receives a "Welcome to Salesforce" email message with a link to verify her account. After she clicks the link, the `email_verified` value is set to `true`. • They change their email address • They change their password, or a Salesforce admin resets their password • They verify their identity when logging in from a new device or browser • A Salesforce admin creates them as a new user
`first_name`	First name of the queried user.
`last_name`	Last name of the queried user.
`timezone`	Time zone specified in the queried user's settings
`photos`	Map of URLs to the queried user's profile pictures, specified as `picture` or `thumbnail`. Note: Accessing these URLs requires passing an access token. See access token.
`addr_street`	Street specified in the address of the queried user's settings.
`addr_city`	City specified in the address of the queried user's settings.
`addr_state`	State specified in the address of the queried user's settings.
`addr_country`	Country specified in the address of the queried user's settings.

Parameter	Description
`addr_zip`	Zip or postal code specified in the address of the queried user's settings.
`mobile_phone`	Mobile phone number specified in the queried user's settings.
`mobile_phone_verified`	Queried user confirmed that the mobile phone number is valid,
`status`	Queried user's current Chatter status. • `created_date`—`xsd datetime` value of the creation date of the last post by the user, for example, 2010-05-08T05:17:51.000Z. • `body`—Body of the post.
`urls`	Map containing various API endpoints that can be used with the queried user Note: Accessing the REST endpoints requires passing an access token. See access token. • `enterprise` (SOAP) • `metadata` (SOAP) • `partner` (SOAP) • `rest` (REST) • `sobjects` (REST) • `search` (REST) • `query` (REST) • `recent` (REST) • `profile` • `feeds` (Chatter) • `feed-items` (Chatter) • `groups` (Chatter) • `users` (Chatter) • `custom_domain` Note: If the org doesn't have a custom domain configured and propagated, this value is omitted.
`active`	Boolean specifying whether the queried user is active.
`user_type`	Type of the queried user.
`language`	Language of the queried user.
`locale`	Locale of the queried user.
`utcOffset`	Offset from UTC of the queried user's time zone, in milliseconds.
`last_modified_date`	`xsd datetime` format of the last modification of the user, for example, 2010-06-28T20:54:09.000Z.
`is_app_installed`	Value is `true` when the connected app is installed in the user's org, and the user's access token was created using an OAuth flow. If the connected app isn't installed, the response

Parameter	Description
	doesn't contain this value. When parsing the response, check for the existence and value of this property.
`mobile_policy`	Specific values for managing a mobile connected app. These values are available only when the connected app is installed in the current user's org, the app has a defined session timeout value, and the mobile PIN has a length value defined. • `screen_lock`—Length of time to wait to lock the screen after inactivity. • `pin_length`—Length of the identification number required to gain access to the mobile app.
`push_service_type`	Set to `apple` if the connected app is registered with Apple Push Notification Service (APNS) for iOS push notifications. Set to `androidGcm` if it's registered with Google Cloud Messaging (GCM) for Android push notifications. The response value type is an array.

`custom_permissions`	When a request includes the `custom_permissions` scope parameter, the response includes a map containing custom permissions in the org associated with the connected app. If the connected app isn't installed in the org or has no associated custom permissions, the response doesn't contain a `custom_permissions` map.
	Here's an example request.
	`http://MyDomainName.my.salesforce.com/services/oauth2/authorize?response_type=token&client_id=3MVG9lKcPoNINVBKV6EgVJiF.snSDwh62wSSTBcOhHEJKC&redirect_uri=http://www.example.org/qa/security/oauth/useragent_flow_callback.jsp&scope=api%20id%20custom_permissions`
	Here's the JSON block in the identity URL response.
	`"custom_permissions": { "Email.View":true, "Email.Create":false, "Email.Delete":false }`

Setting Custom Login Servers in Android Apps

For special cases—for example, if you're a Salesforce partner using Trialforce—you can redirect your user's login requests to a custom login URI.

In Android, login hosts are known as server connections. You can see the standard list of server connections in the `res/xml/servers.xml` file of the `SalesforceSDK` project. Mobile SDK uses this file to define production and sandbox servers.

For Android, the default login host can potentially be set through any of the following means.

1. MDM enforced
   - At startup, your app's MDM provider configures the login URI.
   - The MDM policy can also hide the navigation bar and Settings icon to prevent users from changing the login host.
2. App configuration through the `servers.xml` file

   You can add your custom servers to the runtime list by creating your own `res/xml/servers.xml` file in your native Android project. The first server listed in your `servers.xml` file is used as the default login server at app startup. The root XML element for `servers.xml` is `<servers>`. This root can contain any number of `<server>` entries. Each `<server>` entry requires two attributes: `name` (an arbitrary human-friendly label) and `url` (the web address of the login server, including the "https://" prefix).

   Here's an example of a `servers.xml` file.

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <servers>
     <server name="XYZ.com Login" url="https://myloginserver.cloudforce.com"/>
   </servers>
   ```

   Note: To test XML changes in an Android emulator, we've found that it's best to:

   a. Force stop the app if it's already running in the emulator.

   b. Uninstall the app in the emulator.

   c. Do a full clean and rebuild.

   d. Run the app.
3. User configuration through the Add Connection button

   Here's how a user can configure a custom login server.

   a. Start the app without logging in.

   b. In the login screen, tap the Settings icon in the upper left corner.

   c. Tap **Change Server**.

   d. Tap **Add Connection**.

   e. To help identify this configuration in future visits, enter a name.

   f. Enter your custom login host's URL. Be sure to include the `https://` prefix. For example, here's how you enter a typical Experience Cloud site URL:

   ```
   https://MyDomainName.my.site.com/fineapps
   ```

   Mobile SDK enables this functionality by default. You can't disable the **Change Server** or **Add Connection** option programmatically in Mobile SDK for Android.

Important:

- In Android, always include the "https://" prefix when specifying the login URL.
- At startup, MDM runtime configuration overrides compile-time settings.

Setting Custom Login Servers in iOS Apps

For special cases—for example, if you're a Salesforce partner using Trialforce—you can redirect your user's login requests to a custom login URI.

In iOS apps, login servers are often called *login hosts*. Mobile SDK defines standard login URIs for production and sandbox servers in the `SalesforceSDKCore` project. These two login hosts appear in the Choose Connection login screen.

For iOS, the default login host can potentially be set through any of the following means.

1. MDM enforced
   - At startup, your app's MDM provider configures the login URI.
   - The MDM policy can also hide the navigation bar and Settings icon to prevent users from changing the login host.
2. App configuration through the `info.plist` file
   - Your app can configure the default login URI in the project's `info.plist` properties file. The login host property name is `SFDCOAuthLoginHost`.
   - At startup, the `SFDCOAuthLoginHost` setting overrides user-defined login hosts.
   - By default, `SFDCOAuthLoginHost` property is set to "login.salesforce.com".
   - Do not use a protocol prefix such as "https://" when specifying the login URI.
3. User configuration through the Add Connection screen

   Here's how a user can configure a custom login server.

   a. Start the app without logging in.

   b. In the login screen, tap the Settings, or "gear," icon  in the top navigation bar.

   c. In the Choose Connection screen, tap the Plus icon  .

   d. (Optional but recommended) To help identify this configuration in future visits, enter a label.

   e. Enter your custom login host's URI. Be sure to omit the `https://` prefix. For example, here's how you enter a typical Experience Cloud site URI:

   ```
   MyDomainName.my.site.com/fineapps
   ```

   Mobile SDK enables this functionality by default. You can disable the Add Connection option by setting `SFLoginHostViewController` properties.

Important:

- At startup, MDM runtime configuration overrides compile-time settings.
- Before version 4.1, Mobile SDK apps for iOS defined their custom login URIs in the app's Settings bundle. In Mobile SDK 4.1 and later, iOS apps lose the Settings bundle. Instead, you can use the `SFDCOAuthLoginHost` property in the app's `info.plist` file to build in a custom login URI.

SEE ALSO:

Customizing the iOS Login Screen Programmatically

Revoking OAuth Tokens

When a user logs out of an app, or the app times out or in other ways becomes invalid, the logged-in users' credentials are cleared from the mobile app. This effectively ends the connection to the server.

Also, Mobile SDK revokes the refresh token from the server as part of logout.

See Revoke Opaque OAuth Tokens

Refresh Token Revocation in Android Native Apps

When a refresh token is revoked by an administrator, the default behavior is to automatically log out the current user. As a result of this behavior:

- Any subsequent REST API calls your app makes will fail.
- The system discards your user's account information and cached offline data.
- The system forces the user to navigate away from your page.
- The user must log into Salesforce again to continue using your app.

These side effects provide a secure response to the administrator's action.

Token Revocation Events

When a token revocation event occurs, the `ClientManager` object sends an Android-style notification. The intent action for this notification is declared in the `ClientManager.ACCESS_TOKEN_REVOKE_INTENT` constant.

`SalesforceActivity.java`, `SalesforceListActivity.java`, `SalesforceExpandableListActivity.java`, and `SalesforceDroidGapActivity.java` implement `ACCESS_TOKEN_REVOKE_INTENT` event listeners. These listeners automatically take logged out users to the login page when the refresh token is revoked. A toast message notifies the user of this occurrence.

Connected Apps

A connected app integrates an application with Salesforce using APIs. Connected apps use standard SAML and OAuth protocols to authenticate, provide single sign-on, and provide tokens for use with Salesforce APIs. In addition to standard OAuth capabilities, connected apps allow Salesforce admins to set various security policies and have explicit control over who can use the corresponding apps.

Here's a general list of information that you provide when you create a connected app.

- Name, description, logo, and contact information
- URL where Salesforce can locate the app for authorization or identification
- Authorization protocol: OAuth, SAML, or both
- IP ranges from where users can log in to connected app (optional)
- Information about mobile policies that the connected app can enforce (optional)

On the most basic level, Salesforce Mobile SDK apps use connected apps to access Salesforce OAuth services, which enable access to Salesforce REST APIs.

About PIN Security

Beginning in version 9.2, Mobile SDK uses the mobile device PIN instead of an app-specific PIN and simplifies the passcode flow.

- For iOS, see About Login and Passcodes on page 65.
- For Android, see Using Passcodes on page 134.

Biometric Authentication

Starting in Mobile SDK 11.0, you can configure your app to use the device system biometric authentication to log in. For example, when the app exceeds its timeout period in the background, the login screen appears upon the user's return to the app. This behavior gives the appearance that the user is logged out, even if the user's login session hasn't expired. The user can then log in using their username and password or their device's biometric authentication if they enabled that option.

Note: Alternatively, you can configure your app to require a passcode after login to achieve a similar effect. This feature is presented as a lock screen, while the biometric authentication introduced in Mobile SDK 11.0 presents a login screen. See also:

- About Login and Passcodes on page 65
- Using Passcodes on page 134

Connected App Configuration

To configure biometric authentication, go to your org's connected app.

1. Add a custom attribute to your connected app with the key `ENABLE_BIOMETRIC_AUTHENTICATION`.
2. To change the timeout period, add the `BIOMETRIC_AUTHENTICATION_TIMEOUT` key. You can adjust its value to the number of minutes that you want the app to be backgrounded for before it locks. If you choose not to add this key, the default value is set for 15 minutes.

Note: The user's authentication token doesn't refresh while the app is locked. We recommend that admins enable a session timeout in the connected app and set the Lock App After attribute to the same value.

API and Customization

To manage biometric opt-in from within the app, use `SalesforceSDKManager` to get the `BiometricAuthenticationManager` instance. You can use this instance to:

- Check whether the user has opted in to biometric authentication.
- Prompt the user to opt in or out of biometric authentication and update the SDK with their response.
- Check whether the app is locked.
- Lock the app immediately.
- Disable the native biometric unlock button entirely.

To opt the user in to biometric authentication, you can use the Mobile SDK-provided prompts or create your own.

To use the Mobile SDK-provided prompts, implement one of these lines.

Android (Kotlin)

```
SalesforceSDKManager.getInstance().biometricAuthenticationManager.presentOptInDialog(fragmentManager)
```

iOS (Swift)

```
SalesforceManager.shared.biometricAuthenticationManager().presentOptInDialog(viewController:
 viewController)
```

If you choose to create a custom prompt, implement one of these lines to pass the user's response to Mobile SDK.

Android (Kotlin)

```
SalesforceSDKManager.getInstance().biometricAuthenticationManager.biometricOptIn(userResponse)
```

iOS (Swift)

```
SalesforceManager.shared.biometricAuthenticationManager().biometricOptIn(optIn:
userResponse)
```

If the user enables biometric authentication, a native button is added to the login screen so that they can trigger the OS prompt.

To create a custom button within the app's web view, you can use an API found in `BiometricAuthenticationManager` to disable the native button. Then, configure the button to redirect to `mobilesdk://biometric/authentication/prompt`. Mobile SDK automatically ignores this redirect and presents the native OS prompt.

Portal Authentication Using OAuth 2.0 and Salesforce Sites

The Salesforce Spring '13 Release adds enhanced flexibility for portal authentication. If your app runs in a Salesforce portal, you can use OAuth 2.0 with a Salesforce site to obtain API access tokens on behalf of portal users. In this configuration you can:

- Authenticate portal users via Auth providers and SAML, rather than a SOAP API `login()` call.
- Avoid handling user credentials in your app.
- Customize the login screen provided by the Salesforce site.

Here's how to get started.

1. Associate a Salesforce site with your portal. The site generates a unique URL for your portal. See Associating a Portal with Salesforce Sites.
2. Create a custom login page on the Salesforce site. See Managing Salesforce Site Login and Registration Settings.
3. Use the unique URL that the site generates as the redirect domain for your users' login requests.

The OAuth 2.0 service recognizes your custom host name and redirects the user to your site login page if the user is not yet authenticated.

Example: For example, rather than redirecting to `https://login.salesforce.com`:

```
https://login.salesforce.com/services/oauth2/authorize?
response_type=code&client_id=<your_client_id>&
redirect_uri=<your_redirect_uri>
```

redirect to your unique Salesforce site URL, such as `https://`*`MyDomainName`*`.my.salesforce-sites.com`:

```
https://MyDomainName.my.salesforce-sites.com/services/oauth2/authorize?
response_type=code&client_id=<your_client_id>&
redirect_uri=<your_redirect_uri>
```

For more information and a demonstration video, see OAuth for Portal Users on the Salesforce Platform Developer Relations Blogs page.

Using MDM with Salesforce Mobile SDK Apps

Mobile Device Management (MDM) can facilitate app configuration, updating, and authentication. Salesforce and Mobile SDK support the use of MDM for connected apps.

To use MDM, you work with a Salesforce administrator and an MDM provider. The Salesforce administrator configures your connected app to suit your use case. The MDM provider is a trusted third party who distributes your mobile app settings to your customers' devices. For example, you can use MDM to configure custom login URLs for your app. You can also use MDM for certificate-based authentication. In this case, you upload certificates to the MDM provider.

MDM enablement does not require changes to your Mobile SDK app code.

The following outline explains the basic MDM runtime flow.

Authentication and Configuration Runtime Flow

1. To download an MDM-enabled Mobile SDK app, a customer first installs the MDM provider's app.
2. The MDM provider uses its app to push the following items to the device:
   - Your Mobile SDK app
   - Any configuration details you've specified, such as custom login URLs or enhanced security settings
   - A user certificate if you're also using MDM for authentication
3. When the customer launches your app, behavior varies according to the mobile operating system.
   - **Android:** If you're supporting for certificate-based authentication, the login server requests a certificate. Android launches a web view and presents a list of one or more available certificates for the customer's selection.
   - **iOS:** The Mobile SDK app checks whether the Salesforce connected app definition enables certificate-based authentication. If so, the app navigates to a Safari window. Safari retrieves the stored MDM certificate and transparently authenticates the device.
4. After it accepts the certificate, the login server sends access and refresh tokens to the app.
5. Salesforce posts a standard screen requesting access to the customer's data.

The following sections describe the MDM configuration options that Mobile SDK supports.

Certificate-Based Authentication

Using certificates to authenticate simplifies provisioning your mobile users, and your day-to-day mobile administration tasks by eliminating usernames and passwords. Salesforce uses X.509 certificates to authenticate users more efficiently, or as a second factor in the login process.

MDM Settings for Certificate-Based Authentication

To enable certificate-based authentication for your mobile users, you need to configure key-value pair assignments through your MDM suite. Here are the supported keys:

Key	Data Type	Platform	Description
`RequireCertAuth`	Boolean	Android, iOS	If true, the certificate-based authentication flow initiates. **Android:** Uses the user certificate on the device for authentication inside a webview. **iOS:** Redirects the user to Safari for all authentication requests.

Key	Data Type	Platform	Description
`ManagedAppCertAlias`	String	Android	Alias of the certificate deployed on the device picked by the application for user authentication. Required for Android only.

Note: There's a minimum device OS version requirement to use certificate-based authentication. For Android, the minimum supported version is 5.0. For iOS, the minimum supported version is 7.0.

Once you save your key-value pair assignments, you can push the mobile app with the updated certificate-based authentication flow to your users via your MDM suite.

Automatic Custom Host Provisioning

You can now push custom login host settings to your mobile users. This spares your mobile users from having to manually type long URLs for login hosts—typically a frustrating and error-prone activity. You can configure key-value pair assignments through your MDM to define multiple custom login hosts for your mobile users.

MDM Settings for Automatic Custom Host Provisioning

To push custom login host configurations to your mobile users, you need to configure key-value pair assignments through your MDM suite. Here are the supported keys:

Key	Data Type	Platform	Description
`AppServiceHosts`	String, String Array	Android, iOS	Login hosts. First value in the array is the default host. **Android:** Requires https:// in the host URL. **iOS:** Doesn't require https:// in the host URL.
`AppServiceHostLabels`	String, String Array	Android, iOS	Labels for the hosts. The number of `AppServiceHostLabels` entries must match the number of `AppServiceHosts` entries.
`OnlyShowAuthorizedHosts`	Boolean	Android, iOS	If true, prevents users from modifying the list of hosts that the Salesforce mobile app can connect to.

Additional Security Enhancements

You can add an extra layer of security for your iOS users by clearing the contents of their clipboard whenever the mobile app is in the background. Users may copy and paste sensitive data as a part of their day-to-day operations, and this enhancement ensures any data they copy onto their clipboards are cleared whenever they background the app.

MDM Settings for More Security Enhancements

To clear the clipboards of your iOS users when the mobile app is in the background, you need to configure key-value pair assignments through your MDM suite. Here is the supported key:

Key	Data Type	Platform	Description
`ClearClipboardOnBackground`	Boolean	iOS	If true, the contents of the iOS clipboard are cleared when the mobile app is backgrounded. This prevents the user from accidentally copying and pasting sensitive data outside of the application.

Note: If the mobile app stops working unexpectedly, the copied data can remain on the clipboard. The contents of the clipboard are cleared once the user starts and backgrounds the mobile app.

This security functionality is available through Android for Android devices running OS 5.0 and greater, and that have Android for Work set up. Contact your MDM provider to configure this functionality for your Android users.

Sample Property List Configuration

Note: Setting key-value pair assignments through a plist is only available on iOS.

One method of setting key-value pair assignments is through an XML property list, or plist. The plist contains the key-value pair assignments that an MDM provider sends to a mobile app to enforce security configurations.

Here is a sample plist:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
 <key>AppServiceHosts</key>
 <array>
  <string>host1</string>
  <string>host2</string>
 </array>
 <key>AppServiceHostLabels</key>
 <array>
  <string>Production</string>
  <string>Sandbox</string>
 </array>
 <key>RequireCertAuth</key>
 <true/>
```

```
 <key>ClearClipboardOnBackground</key>
 <false/>
 <key>OnlyShowAuthorizedHosts</key>
 <false/>
</dict>
</plist>
```

Using Advanced Authentication

By default, Mobile SDK automatically uses standard authentication. On the server side, however, Salesforce orgs can choose to use advanced authentication by configuring either My Domain or MDM. Advanced auth requires a small amount of configuration in most Mobile SDK apps.

Which Type of Auth Will Mobile SDK Use?

At runtime, Mobile SDK bases its authentication type on the login org's configuration.

- If browser-based authentication is configured for the org's My Domain, Mobile SDK conforms to the My Domain setting.
- If browser-based authentication hasn't been configured for the org's My Domain, Mobile SDK uses advanced auth only if the org uses MDM certificate-based auth.
- If the org doesn't use My Domain browser-based authentication or MDM certificate-based auth, Mobile SDK uses standard auth.

Advanced Auth User Flow on iOS

For browser-based authentication, customers log in through the familiar Salesforce web view, followed by an authorization screen.

For certificate-based authentication, instead of a login screen, the operating system prompts the customer to choose a certificate for authentication. The customer does not enter credentials. After choosing the certificate, the customer sees the authorization screen.

Development Requirements

Mobile SDK requirements for implementing advanced auth are minimal. Most apps require only a small amount of configuration. Android apps that use MDM certificate-based auth do not require client-side configuration.

Configuring a Connected App

- In a Salesforce connected app, under API (Enable OAuth Settings):
  - Apply the typical OAuth settings for Mobile SDK apps. See API (Enable OAuth Settings).
  - Make sure that **Require Secret for Web Server Flow** is *not* selected.

Configuring My Domain Settings

An org administrator can require advanced auth through My Domain settings. To take advantage of advanced auth:

1. From Setup, in the Quick Find box, enter `My Domain`, and then select **My Domain**.
2. In My Domain settings, under Authentication Configuration, the administrator selects one or both of the following options:
   - **Use the native browser for user authentication on Android**

- **Use the native browser for user authentication on iOS**

See "Customize Your My Domain Login Page for Mobile Auth Methods" in Salesforce Help.

Configuring MDM Settings

For logins managed through MDM, Mobile SDK uses advanced auth only if the org's MDM settings specify certificate-based auth. An org's MDM suite must:

- Set the `RequireCertAuth` property to `true`.
- **Android only:** Set the `ManagedAppCertAlias` property to an alias name.

Login Session Management with Advanced Authentication

With advanced auth, logging out of an app can cause surprising behavior. How this behavior can affect your app depends on the type of login your app uses.

Certificate-Based (MDM) Login

With certificates, a customer remains authenticated until the certificate is revoked. A certificate remains valid until the issuer revokes it. If a customer logs out of the app while the certificate is valid, the Salesforce login screen appears briefly. However, because the certificate automatically supplies the customer's credentials, the flow goes directly to the authorization ("Allow Access") screen. By choosing **Allow**, the customer obtains new access and refresh tokens and can continue using the app. In effect, a customer can't log out until the MDM issuer revokes the certificate.

Web Server OAuth Login

During OAuth 2.0 authentication, Salesforce creates a temporary short-term session to bridge the gap between login and the Salesforce authorization ("Allow Access") screen. This temporary session, which uses a cookie, is not tied to the OAuth refresh or access token and therefore isn't invalidated at logout. Instead, the session remains valid until it expires. The most recently authenticated customer remains logged in until the temporary session expires. These sessions have an intentionally short lifetime, after which the user can log in normally.

The following unexpected behavior can occur during the lifetime of the temporary session: If the customer tries to log out and log in again before the cookie expires, the flow skips the login prompt. Instead, it goes directly to Salesforce authorization. By choosing **Allow**, the customer automatically obtains new access and refresh tokens and can continue using the app.

This behavior doesn't occur with standard web view authentication because the web view doesn't preserve cookies from previous authentications. It also doesn't occur if the customer logs out after the temporary session expires. Mobile SDK apps, including the Salesforce app, can't control cookies from the Salesforce service.

Important: Although advanced auth doesn't use swizzling, the login page remains full-screen. This presentation can give customers the impression that they've temporarily left your app.

See Also

- Using MDM with Salesforce Mobile SDK Apps.
- For information on server-side My Domain configuration, see Customize Your My Domain Login Page with Your Brand in *Salesforce Help*.
- For connected app details, see Create a Connected App in *Salesforce Help*.
- For MDM configuration details, see "Mobile Device Management (MDM)" in *Salesforce Mobile App Security Guide*.
- For information on configuring iOS URL schemes, look up at "Inter-App Communication" or "Custom URL Schemes" in the *App Programming Guide for iOS* at developer.apple.com/documentation.

IN THIS SECTION:

Configuring Advanced Authentication in iOS Apps

To support advanced auth, all iOS apps require some custom configuration.

Advanced auth in iOS uses the latest iOS technology supported by the current Mobile SDK release.

Standard Authentication Versus Advanced Authentication

Here's a partial list of differences between standard and advanced auth on iOS. These differences are specific to My Domain browser-based authentication.

- *Standard auth flow*: This flow uses `WKWebView`. This class offers a superior user experience with access to the iOS view toolbar and other compelling features.
- *Advanced auth flow*: Advanced auth uses the latest iOS technology supported by the current Mobile SDK release. It's the more secure option—it doesn't allow the app to set cookies or inject content into the view without the customer's consent. In advanced mode, the auth flow doesn't swizzle.

App Configuration

In iOS apps, the steps are the same for both MDM certificate-based and browser-based approaches. Perform the following steps to guarantee compatibility with all orgs.

- Add your custom URL schemes for the OAuth redirect URI to your project's `Info.plist` file.
  1. In your app's `Info.plist` file, create a key named `CFBundleURLTypes`.
  2. Assign the key an array that contains a dictionary with the following keys:

Name	Type	Value
`CFBundleURLName`	String	A unique abstract name of the URL scheme, preferably an identifier in reverse-DNS style. For example: `com.acme.myscheme` .
`CFBundleURLSchemes`	Array of strings	URL scheme names, such as `http` and `mailto`.

Example: If your OAuth callback URI is `com.mydomain.myapp://oauth/success`, add the following key to your `Info.plist` file:

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleURLName</key>
        <string>com.mydomain.myapp</string>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>com.mydomain.myapp</string>
        </array>
    </dict>
</array>
```

In this example, the URL scheme and URL name are the same, but this matching is not required. You can add as many schemes as your app requires.

See Also

- For information on server-side My Domain configuration, see Customize Your My Domain Login Page with Your Brand in *Salesforce Help*.
- For MDM configuration details, see "Mobile Device Management (MDM)" in *Salesforce Mobile App Security Guide*.
- For information on configuring iOS URL schemes, look up at "Inter-App Communication" or "Custom URL Schemes" in the *App Programming Guide for iOS* at developer.apple.com/documentation/..

Configuring Advanced Authentication in Android Apps

In Salesforce orgs that use My Domain for advanced authentication, Mobile SDK requires a small amount of configuration in the client app. Android apps that use certificate-based authentication don't require configuration within the Mobile SDK app.

Android Implementation

For advanced authentication support in Android applications, Mobile SDK uses a Chrome custom tab. If Chrome isn't available at runtime, Mobile SDK uses the default system browser. Browser-based authentication requires the following.

- A browser must be installed on the device.
- If you use MDM, the browser must be installed in the work partition.

Optionally, you can configure which browser the application selects by using this method.

```
SalesforceSDKManager.getInstance().setCustomTabBrowser(browserPackage);
```

To see the currently selected custom tab browser, use this method.

```
browserPackage = SalesforceSDKManager.getInstance().getCustomTabBrowser();
```

Certificated-Based App Configuration

Certificate-based authentication relies on an MDM vendor. This vendor brokers identification services between Salesforce and the client mobile device. Certificate-based authentication doesn't require configuration in Mobile SDK Android projects.

Browser-Based App Configuration

1. In Android Studio, open your app's `AndroidManifest.xml` file.
2. In the `LoginActivity` declaration, uncomment the following lines:

```
<activity android:name="com.salesforce.androidsdk.ui.LoginActivity"
    android:theme="@style/SalesforceSDK.ActionBarTheme"
    android:launchMode="singleInstance">

    <!--
    <intent-filter>
        <data android:scheme="testsfdc"
            android:host="*"
            android:path="/mobilesdk/detect/oauth/done" />
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.BROWSABLE" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
    -->
</activity>
```

3. Replace the values for `android:scheme`, `android:host`, and `android:path` with their corresponding values from your connected app. Here's a couple of examples.

   If the callback URL of your connected app is `testsfdc:///mobilesdk/detect/oauth/done`:

   - `android:scheme` is `testsfdc`.
   - `android:host` is `*`, meaning that it doesn't exist.
   - `android:path` is `/mobilesdk/detect/oauth/done`.

   If the callback URL of your connected app is `sfdc://login.salesforce.com/oauth/done`:

   - `android:scheme` is `sfdc`.
   - `android:host` is `login.salesforce.com`.
   - `android:path` is `/oauth/done`.

Here's the updated portion of your `AndroidManifest.xml`, using the `testsfdc:///mobilesdk/detect/oauth/done` scheme.

```
<!-- Login activity -->
  <!--
     Launch mode of "singleInstance" ensures that the activity isn't restarted
     by a callback from Chrome custom tab when auth flow is complete. This is
     required for the Chrome custom tab auth flow to work correctly.
  -->

  <!--
     To enable browser bath authentication, uncomment the lines below and replace
     'scheme', 'host' and 'path' with their corresponding values from your connected app.


     For example, if the callback URL of your connected app is
     "testsfdc:///mobilesdk/detect/oauth/done",
     'scheme' would be "testsfdc", 'host' would be "*" since it doesn't exist, and
```

```
     'path' would be "/mobilesdk/detect/oauth/done".

     If the callback URL is "sfdc://login.salesforce.com/oauth/done",
     'scheme' would be "sfdc", 'host' would be "login.salesforce.com",
     and 'path' would be "/oauth/done".
  -->
<activity android:name="com.salesforce.androidsdk.ui.LoginActivity"
  android:theme="@style/SalesforceSDK.ActionBarTheme"
  android:launchMode="singleInstance">
  <intent-filter>
    <data android:scheme="testsfdc"
      android:host="*"
      android:path="/mobilesdk/detect/oauth/done" />
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.BROWSABLE" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

You're all set!

See Also

- Customize Your My Domain Login Page with Your Brand in *Salesforce Help*.
- "Mobile Device Management (MDM)" in *Salesforce Mobile App Security Guide*.
- For information on configuring iOS URL schemes, see "Inter-App Communication" or "Custom URL Schemes" in the *App Programming Guide for iOS* at developer.apple.com/documentation/..

Upgrading Android Single Sign-On Apps to Google Login Requirements

In 2018, Google officially dropped support for logins through embedded web views in favor of browser-based login flows. As a result, older Mobile SDK Android apps that use Google as an SSO identity provider might require code changes.

Browser-based authentication is also known as "advanced authentication". To upgrade your older apps, you implement advanced authentication as described in Configuring Advanced Authentication in iOS Apps.

Mobile SDK supports advanced authentication through a Chrome custom tab in the application. If Chrome is not available at runtime, Mobile SDK uses the default system browser. Browser-based authentication requires the following.

- A browser must be installed on the device.
- If you use MDM, the browser must be installed in the work partition.

See Also

- Chrome Custom Tabs
- Google's developer blog

Using OpenID Tokens to Access External Services

If your Mobile SDK app requires approved services external to Salesforce, you can use OpenID tokens to perform the necessary authentication handshake.

Most Mobile SDK apps authenticate with Salesforce and use Mobile SDK REST API wrappers to access Salesforce resources. In this scenario, Mobile SDK handles authentication token exchanges behind the scenes without the app's explicit involvement.

Some apps, however, also require data from an authenticated service that doesn't accept Salesforce tokens. These services can come from products owned by Salesforce, such as Quip or Heroku, or sanctioned third parties. To make these external API calls from the Salesforce environment, you can use OpenID tokens.

OpenID Tokens

An OpenID token allows the app that generates the token to share information with an external web service. For Mobile SDK purposes, the OpenID token shares the user's and app's identities. The external service that receives this token can then give the app a full set of external credentials for the user. Typically, OpenID tokens provided by Salesforce have short lifespans to limit opportunities for security breaches.

OpenID support requires configuration on the Salesforce server and in the Mobile SDK app. On the Salesforce side, org administrators configure connected apps to support OpenID tokens. In the Mobile SDK app, the developer configures the app's OAuth scopes and calls a Mobile SDK method that provides an OpenID token. The app can then exchange this token for a full set of credentials from the external service. **The app is responsible for managing any external credentials it uses.**

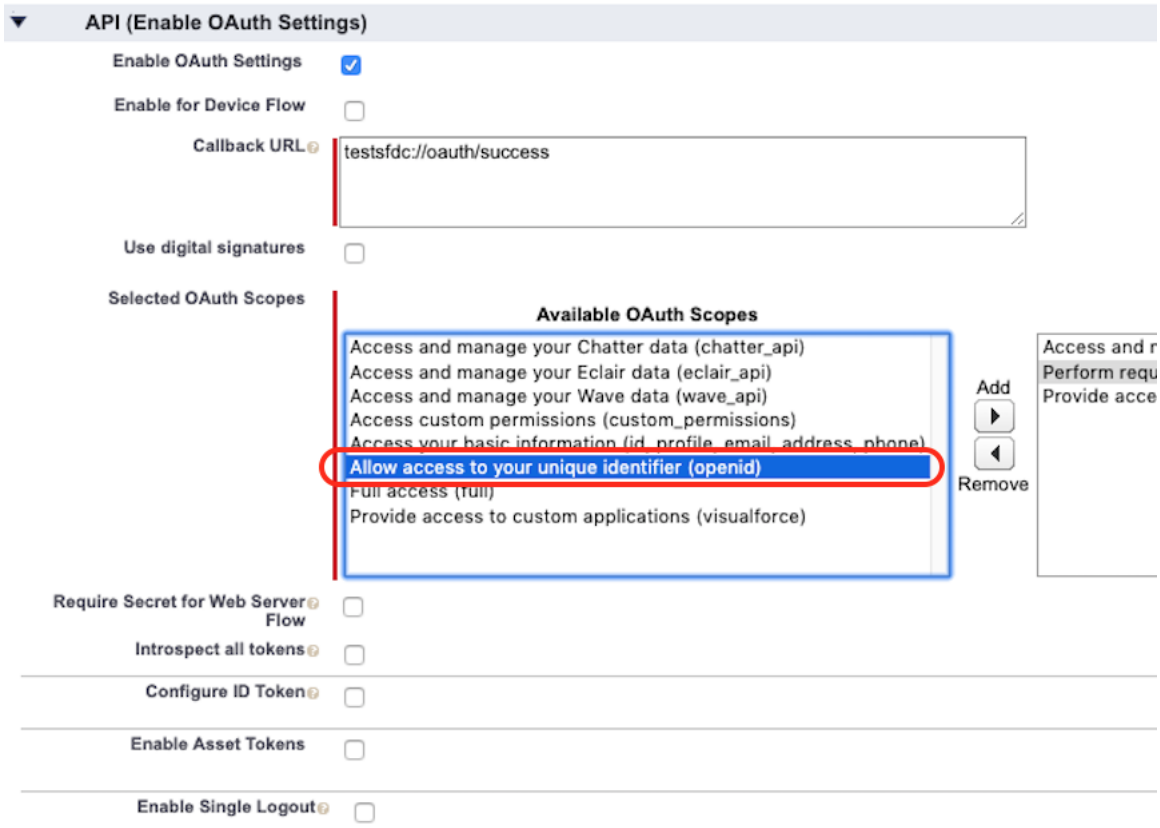
For more information on OpenID, see openid.net/what-is-openid.

Configure Server-Side Settings

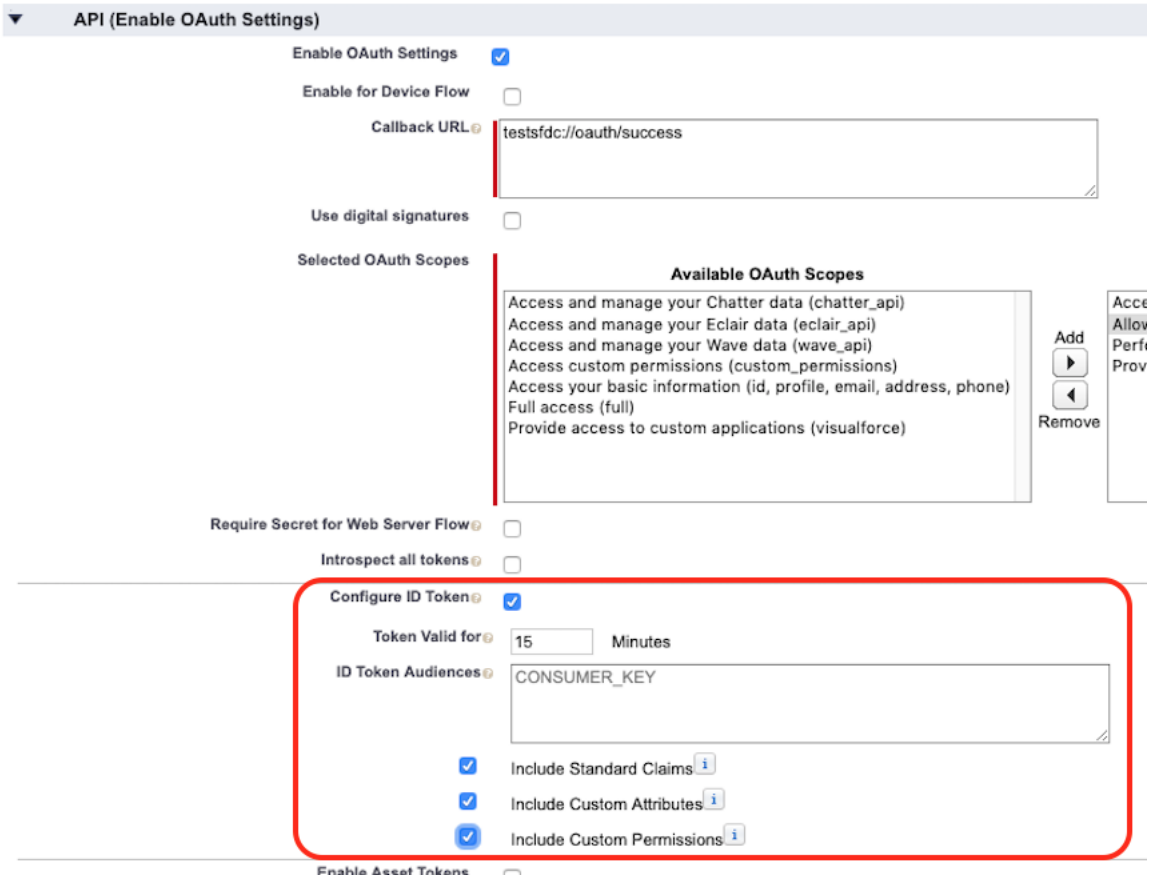
Connected app settings under API (Enable OAuth Settings) when you edit a new or existing connected app.

1. Select **Enable OAuth Settings**.

2. Under Selected OAuth Scopes, select **Allow access to your unique identifier (openid)** and click **Add**.



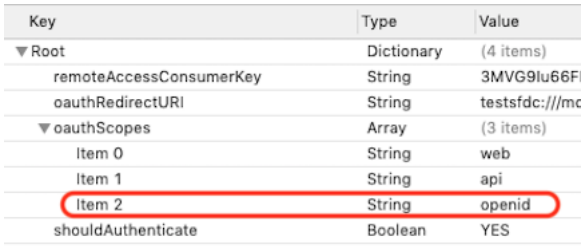
3. Select **Configure ID Token** and configure its subsettings as described in Create a Connected App in *Salesforce Help*.



App Configuration

1. In the `bootconfig.xml` file (Android) or the `bootconfig.plist` file (iOS), add `openid` to the `oauthScopes` list.

   iOS:



Key	Type	Value
▼ Root	Dictionary	(4 items)
remoteAccessConsumerKey	String	3MVG9Iu66F
oauthRedirectURI	String	testsfdc:///mc
▼ oauthScopes	Array	(3 items)
Item 0	String	web
Item 1	String	api
Item 2	String	openid
shouldAuthenticate	Boolean	YES

   Android:

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
    <string name="remoteAccessConsumerKey">3MVG9Iu66FKeHhINkB1
    <string name="oauthRedirectURI">testsfdc:///mobilesdk/dete
    <string-array name="oauthScopes">
        <item>api</item>
        <item>web</item>
        <item>openid</item>
    </string-array>
    <string name="androidPushNotificationClientId"></string>
</resources>
```

2. To obtain an OpenID token string, call the platform-specific API.

   iOS (Objective-C)

   Call the following method on the `SFSDKOAuth2` class.

   ```
   - (void)openIDTokenForRefresh:(SFSDKOAuthTokenEndpointRequest *)endpointReq
                      completion:(void (^)(NSString *))completionBlock;
   ```

   Android

   Call the following method on the `OAuth2` class.

   ```
   public static String getOpenIDToken(String loginServer, String clientId,
       String refreshToken);
   ```

See Also

- Create a Connected App in *Salesforce Help*.

Secure Key Storage in iOS

To protect customer information, Mobile SDK encrypts sensitive data such as user identity tokens and SmartStore databases. In a normal workflow, the data protection pattern Mobile SDK uses is considered highly secure. However, "zero day" vulnerabilities can arise even in the most heavily guarded schemes. To stay ahead of hackers and malicious attacks, Mobile SDK continually upgrades its key encryption to the highest standards iOS supports.

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

In iOS, each type of customer data uses its own unique key. These keys are stored in the iOS keychain and encrypted by a master key.

Within the iOS keychain, the master key is encrypted on disk and is accessible only to the app. The master key is secured with an industry-standard 256-bit Advanced Encryption Standard (AES) key. Nevertheless, a slight risk of exposure to malicious apps still exists. To impose extra protection, Mobile SDK provides the following enhancements.

- It stores the master key in Apple's hardware-based Secure Enclave. Mobile SDK apps never access private keys stored in the Secure Enclave. To create keys, store them securely, and perform other protected operations, Mobile SDK itself calls Secure Enclave APIs. The app receives only the requested output, such as encrypted SmartStore data, without handling unencrypted sensitive data.
- It protects the master key with a 256-bit elliptic curve cryptography (ECC) private key. A 256-bit ECC private key is equivalent to a 3072-bit RSA private key. ECC format is the basis of cryptocurrencies such as BitCoin and Ethereum.

When an upgraded app first runs on a device, Mobile SDK automatically converts the master key to the current encryption level and moves it to the Secure Enclave.

Note: Secure Enclave is available only on devices with Apple A7 or later A-series processors. On devices that don't support Secure Enclave, the master key remains in the keychain.

Upgrading Encryption in Apps

Mobile SDK 9.2 updates its default encryption from AES-CBC to AES-GCM. For most apps, Mobile SDK handles this upgrade silently without requiring app changes. For a few cases, though, the app itself must perform a minor upgrade step.

Action on your part is required if:

- **Your app initializes `KeyValueEncryptedFileStore` directly rather than going through the shared store class methods.**

  In this case, before initializing the key store, call **`KeyValueEncryptedFileStore.updateEncryption(_:_:_:)`**. For the `legacyKey` argument, pass the key that was used originally to create the store. After the upgrade, the key will be managed by Mobile SDK. For example:

```
// Pre 9.2
let key = SFKeyStoreManager.sharedInstance().retrieveKey(withLabel: "kv_key",
    autoCreate: true)
let store = KeyValueEncryptedFileStore(parentDirectory: "directory/path",
    name: "storeName", encryptionKey: key)

// 9.2 upgrade
let key = SFKeyStoreManager.sharedInstance().retrieveKey(withLabel: "kv_key",
    autoCreate: true)
KeyValueEncryptedFileStore.updateEncryption(parentDirectory: "directory/path",
    name: "storeName", legacyKey: key)
let store = KeyValueEncryptedFileStore(parentDirectory: "directory/path",
    name: "storeName")
```

- **Your app uses the `SFSmartStore setEncryptionKeyBlock:` Objective-C method.**

  To upgrade store encryption, continue using `setEncryptionKeyBlock:` with the original key. Mobile SDK 9.2 or later performs the encryption upgrade at runtime and then replaces the default SmartStore key with a new key that it generates.

  Although this automatic key replacement is the recommended path, you can opt to replace the default key yourself. To do so, call `setEncryptionKeyGenerator:` with a new key that you provide.

  In a future release, you can remove your call to `setEncryptionKeyBlock:`.

See Also

- Storing Keys in the Secure Enclave (developer.apple.com/documentation/security)
- Elliptic Curve Cryptography (wikipedia.com)

Secure Key Storage in Android

Mobile SDK encrypts data such as user identity tokens and SmartStore databases. In a normal workflow, the data protection pattern Mobile SDK uses is considered highly secure. However, "zero day" vulnerabilities can arise even in the most heavily guarded schemes. To stay ahead of hackers and malicious attacks, Mobile SDK reinforces its encryption with the highest standards Android supports.

For some time, Mobile SDK has used a symmetric key pair for its encryption tasks. Mobile SDK 7.1 and later adds another level of encryption on top of this scheme that takes advantage of the Android Keystore. Any device that meets the current Mobile SDK Android requirements can support Android Keystore. Keystore implementation is hardware-dependent and varies with Android API version, device manufacturer, and other factors.

To enhance security, Mobile SDK generates an asymmetric public-private key pair to encrypt its symmetric key pair. This asymmetric key pair, which uses RSA-2048 encryption, is stored in the Android Keystore. At runtime, Mobile SDK encrypts the symmetric key with the asymmetric public key and then stores the encrypted key in a SharedPreferences file. To decrypt customer data, the app asks Mobile SDK for the symmetric key. To access that key, Mobile SDK fetches the asymmetric private key from the Keystore and uses it to decrypt the contents of the SharedPreferences file. Mobile SDK then delivers the unencrypted symmetric key to the application.

Upgrading Apps

Mobile SDK automatically upgrades its keys to the new encryption scheme. Behavior and usage of `getEncryptionKey()` is unchanged.

See Also

- "Android keystore system" at developer.android.com

CHAPTER 15 Login Screen Customization

In this chapter ...

Although Mobile SDK doesn't control the Salesforce login page, you can still customize and brand it in certain cases.

Customize the Login Screen Appearance through Mobile SDK

On iOS, you can also configure some properties of the login page, including the navigation bar and Settings icon. See Customizing the iOS Login Screen Programmatically

Customizing the Login Screen through the Salesforce Server

Salesforce Mobile SDK provides an OAuth implementation for its client apps, but it doesn't define or control the login page. Instead, it requests the page from the Salesforce server. Salesforce itself then presents a web view that gathers your customer's credentials. The login web view is not part of your Mobile SDK app.

To change the login web view on the server side, use either your My Domain login URL or an Experience Cloud site.

Both of these features provide handy utilities for login page branding and customization. To use your branded page, you set the default login URL of your Mobile SDK app to the Experience Cloud site or My Domain login URL. Your app then displays your customized login page.

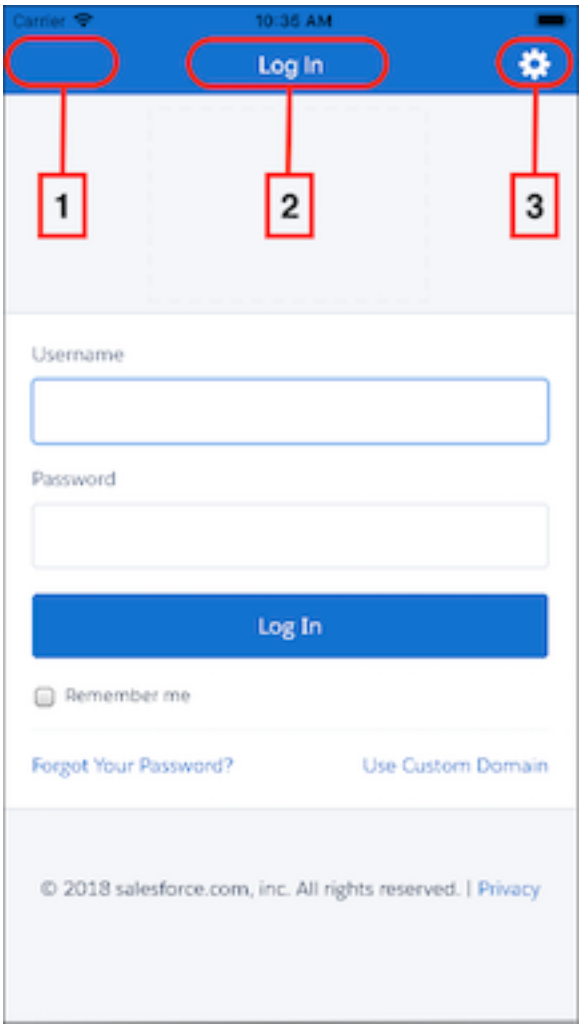
Use the following links to learn about these features.

- Use Your Branded Login Page
- Customize Login, Self-Registration, and Password Management for Your Experience Cloud Site
- My Domain
- Customize Your My Domain Login Page with Your Brand
- Sample My Domain customized login page—https://github.com/salesforceidentity/MyDomain-Sample

Customizing the iOS Login Screen Programmatically

Mobile SDK for iOS provides extensive options for customizing the style and behavior of the login screen's navigation bar. You can make simple declarative changes to control widget appearance and visibility, or you can reimagine the navigation bar by extending the login view controller class.

Here's the standard Salesforce login screen.



Navigation bar widgets you can customize include the back button (1), which is normally hidden, the title (2), and the Settings icon (3). You can also hide the entire navigation bar. By default, the login screen shows both the top navigation bar and its embedded Settings icon.

The Settings icon is often referred to as the “gear” icon due to its sprocket-like shape. Customers can use the Settings icon to select a login server from a built-in list, or to specify a custom login URI. Some companies, however, don’t allow users to choose the login server. To disable login server selection, you can hide either the Settings icon itself or the entire navigation bar. You can also customize the navigation bar's color and the color and font of its text. Use the following `SFSDKLoginViewControllerConfig` properties to control the visibility and appearance of these UI elements.

showSettingsIcon

Controls the display of the Settings icon only. Does not affect the visibility of the navigation bar.

Behavior

Value	Meaning
`YES` (default)	Default value. The Settings icon is visible and accessible.
`NO`	The Settings icon is hidden. Users cannot access the login host list and cannot add custom hosts.

showNavbar

Controls the display of the navigation bar, which in turn hides the Settings icon.

Behavior

Value	Meaning
`YES` (default)	Default value. The navigation bar is visible. The Settings icon display depends on the `showSettingsIcon` property.
`NO`	The navigation bar and the Settings icon are hidden. Users cannot access the login host list and cannot add custom hosts.

Note: To hide the gear icon in hybrid apps, apply these steps in the native wrapper app.

Navigation Bar Colors and Font

You can set the following style properties:

- `navBarColor`
- `navBarTextColor`
- `navBarFont`

The following example shows how to set the Settings icon visibility and navigation bar style. You import the header file and add the subsequent lines to the `application:didFinishLaunchingWithOptions:` method of your `AppDelegate` class.

```
let loginViewConfig = SalesforceLoginViewControllerConfig()
loginViewConfig.showsSettingsIcon = false
loginViewConfig.showsNavigationBar = true
loginViewConfig.navigationBarColor = UIColor(red: 0.051, green: 0.765, blue: 0.733, alpha:
 1.0)
loginViewConfig.navigationBarTextColor = UIColor.white
loginViewConfig.navigationBarFont = UIFont(name: "Helvetica", size: 16.0)
UserAccountManager.shared.loginViewControllerConfig = loginViewConfig
```

Overriding Navigation Bar Widgets by Extending SFLoginViewController

To provide in-depth customization of the navigation bar, extend the `SFLoginViewController` class and implement the required methods. Doing so gives you the power to

- **Enable the back button**—Some developers enable the back button to enhance the customer experience if login fails. You're responsible for providing a custom action if you enable the back button. If you don't customize the action, Mobile SDK uses its default behavior for failed logins, which ignores the back button.
- **Override the default title widget**—You can provide your own title text and define custom actions.
- **Override the default Settings icon**—You can substitute a custom icon for the gear image.

Here's a partially coded example.

1. Extend `SFLoginViewController`.

```
@interface SFLoginExtendedViewController : SFLoginViewController

@end

@implementation SFLoginExtendedViewController

- (UIBarButtonItem *)createBackButton {
    // Setup left bar button:
    // UIImage *image = [[SFSDKResourceUtils
    //                 imageNamed:@"globalheader-back-arrow"]
    //    imageWithRenderingMode:UIImageRenderingModeAlwaysTemplate];
    //
    // Return a custom UIBarButtonItem:
    // return [[UIBarButtonItem alloc]
    //    initWithImage:image
    //            style:UIBarButtonItemStylePlain
    //           target:self action:nil];
}

- (void)handleBackButtonAction {
    [super handleBackButtonAction];
    // Add your custom code here
}

- (BOOL)shouldShowBackButton {
    // Add your custom code here.
    // Return YES to show the back button
}

- (UIBarButtonItem *)createSettingsButton {
    // Set up left bar button:
    // UIImage *image = [[SFSDKResourceUtils
    //                  imageNamed:@"login-window-gear"]
    //     imageWithRenderingMode:UIImageRenderingModeAlwaysTemplate];
    //
    // Return a custom UIBarButtonItem:
    // return [[UIBarButtonItem alloc]
    //     initWithImage:image
    //             style:UIBarButtonItemStylePlain
    //            target:self action:nil];
```

```
}

- (UIView *)createTitleItem {
    // Setup top item.
    // Create a UIView for title
    // UIView *item = [[UIView alloc] ... ];
    NSString *title = [SFSDKResourceUtils localizedString:@"MobileSyncLogin"];
    ...
    return item;
}

- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view.
}

@end
```

2. In the `application:didFinishLoadingWithOptions:` method of your `AppDelegate` class, set up the following block.

```
...
[SFUserAccountManager sharedInstance].
    loginViewControllerConfig.loginViewControllerCreationBlock =
        ^SFLoginViewController * _Nonnull{
            SFLoginExtendedViewController *controller =
                [[SFLoginExtendedViewController alloc] init];
            return controller;
    };
[SFUserAccountManager sharedInstance].loginViewControllerConfig =
    loginViewConfig;
....
```

SEE ALSO:

Setting Custom Login Servers in iOS Apps

CHAPTER 16 Identity Provider Apps

In this chapter ...

Identity providers help known users avoid reentering their Salesforce credentials every time they log in to a Mobile SDK app. At the same time, it preserves the stringent security level of previous Mobile SDK releases.

An identity provider setup consists of two primary components:

- An identity provider (IDP) is an ordinary Mobile SDK app that's configured to manage Salesforce logins for one or more users on a single mobile device. This app serves as the broker between Mobile SDK apps on the device and the Salesforce authentication service. It tracks device users that have recently logged in and kicks off the authentication process when they return to the app.
- An identity provider client is an ordinary Mobile SDK app that's configured to use an identity provider for logins. These apps are also called "service providers" or "SPs" because they provide the services that the user is trying to access. A traditional service provider gives the user one choice for authentication: the Salesforce login screen. An identity provider client, on the other hand, gives the user the choice of logging in through either the Salesforce login screen or a specific identity provider. With the identity provider option, an authorization request to the current active user in the IDP app is sent to a Salesforce authorization endpoint.

Typically, a single device hosts a single identity provider app, but this limitation is not enforced. A single device can host any number of identity provider client apps.

Note: Mobile SDK introduced support for identity provider services in version 6.0. In Mobile SDK 11.0, we reworked our support to improve the user experience and added new functionality. Apps built with Mobile SDK versions 6.0–10.2 can continue to use the IDP / SP flows introduced in Mobile SDK 6.0. However, apps built on Mobile SDK 11.0 and on require the respective IDP/ SP flows reworked in Mobile SDK 11.0 to implement the feature.

Several rules apply to identity provider apps and their clients. The following apply to the latest IDP and SP flows supported in Mobile SDK 11.0.

- They must be built with Mobile SDK 11.0 or later.
- They must be discrete apps. You can't combine identity provider and identity provider client implementations in a single app.
- A client and the identity provider it delegates to must use different Salesforce connected apps with differing OAuth callback URIs.

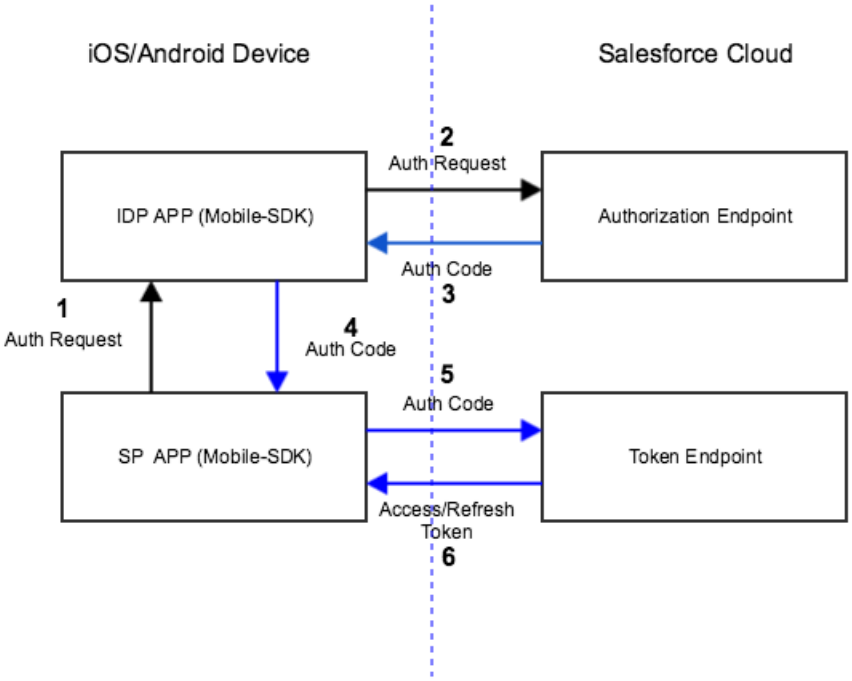
Identity Providers: Architecture, Flow, and Connected App Requirements

All Mobile SDK identity provider setups follow the same flows and architectural outlines, regardless of platform. Salesforce connected apps for identity provider components also impose a few special requirements.

High-Level Flow

A user can start the identity provider flow from either an identity provider app or one of its clients. When a user starts the flow by launching an identity provider client app, the first step is to choose a login preference: through either the standard Salesforce login view or an identity provider. The user can also choose identity provider login by directly launching an identity provider app. In this case, the identity provider launches the requested client app to start authentication.

Here's a high-level diagram of what happens when the flow begins in an identity provider client. The client app doesn't collect credentials from the user. Instead, it delegates login interaction to the identity provider app.



1. At the user's request, the identity provider client app requests authorization from an identity provider. The identity provider responds by sending an authorization request to a Salesforce authorization endpoint.
2. Salesforce sends an authorization code back to the identity provider.
3. Identity provider forwards the authorization code to the identity provider client.
4. Identity provider client forwards the authorization code to the Salesforce OAuth token endpoint.
5. Salesforce exchanges the authorization code for access and refresh tokens and returns the tokens to the identity provider client app. Only the client app handles the user's OAuth tokens.

All the user has to do is click the identity provider button, then select an account. After accepting the standard request for access to Salesforce data, the user can begin working in the client app.

A Bit More Detail

Here's more granular information for the authentication flow. In the identity provider client:

1. If the user has a valid refresh token in the identity provider client app, they're automatically in––just the usual refresh token exchange occurs.
2. If the user doesn't have a valid refresh token, the identity provider client prompts the user to choose either standard Salesforce login or identity provider login.

Mobile SDK ignores a user's selection of identity provider login and instead displays the standard Salesforce login screen if:

- No identity provider app is configured in the identity provider client app.
- The configured identity provider app is not found.

If authorization succeeds, the client asks Salesforce to refresh the user's tokens. When Salesforce returns the refresh token, the flow defaults to the standard Mobile SDK post-login flow. Mobile SDK caches the refresh token returned from Salesforce.

Connected App Requirements

- Create two connected apps––one for the identity provider app, one for identity provider client apps.
- Uncheck "Require Consumer Secret" on the connected app for identity provider client apps.
- Set the OAuth Callback URI to a custom scheme that you devise. The custom URI scheme for an identity provider client app can't match the scheme for its identity provider app.

Important:

- An identity provider app must enable the `web` scope on the connected app and request it in the mobile app. Otherwise, Mobile SDK redirects the user to a login page with an error message. At this point, the flow can't continue.
- An identity provider and its clients must use different connected apps with their own `consumerKey` and `callbackUrl` values. Otherwise, the Salesforce service returns an "invalid credentials" error message when the identity provider requests an authorization code. Currently, all Mobile SDK sample apps use the same `consumerKey` and `callbackUrl`. If you plan to adopt these apps as identity provider-client pairs, make sure that each app uses its own unique values.

Android Architecture and Flow

To kick off the identity provider flow, a user can open either an identity provider or an identity provider client app. In Android, the implementation differs depending on which side initiates it.

Identity Provider Client

The following steps describe the identity provider flow when launched from an identity provider client.

1. User launches the identity provider client and chooses "Log in with IDP".
2. Identity provider client swizzles to the identity provider for authentication.
3. Identity provider completes authentication and swizzles back to the client app.

Initiated by an Identity Provider

The following steps describe the identity provider flow when launched from an identity provider.

1. User launches the identity provider app.
2. User selects a client app.
3. An authorization request for the current active user in the IDP app is sent to a Salesforce authorization endpoint.
4. Identity provider app communicates with the selected client app in the background, passing the username (`user_hint`). If the client app has the user, it tells the user that it's ready to go, and the flow continues to step 6.
5. If the client app doesn't have the user:
   - Client app responds back to identity provider app to request an authentication code.
   - Identity provider app gets the authentication code and sends it to the client app.
   - Client app performs the refresh token exchange for the authentication code and communicates to the IDP app that it's ready to go.
6. The identity provider app swizzles to the client app, which is now logged in.

Configuring an Android App as an Identity Provider

You can configure any app built on Mobile SDK 11.0 or later as an identity provider. You call a method to define which identity provider client apps you want to connect to, then select the identity provider client in your app's UI.

The easiest way to create an identity provider app is by using the Mobile SDK AndroidIDPTemplate. This template is available on GitHub in the github.com/forcedotcom/SalesforceMobileSDK-Templates repo. Use the *`forcedroid createwithtemplate`* command with the URI of the template repo, as shown in the following command-line example.

```
$ forcedroid createwithtemplate
Enter URI of repo containing template application: AndroidIDPTemplate
Enter your application name: MyIDP-Android
Enter your package name: com.acme.android
Enter your organization name (Acme, Inc.): Acme Systems
Enter output directory for your app (leave empty for the current directory): MyIDP-Android
```

Convert an Existing Mobile SDK Android App into an Identity Provider

To let the identity provider app know about the client app you want it to service, define one (or multiple) client app configurations in the `onCreate` method of its application subclass.

```
class MyApplication : Application() {

 override fun onCreate() {
   super.onCreate()
   SalesforceSDKManager.initNative(applicationContext, MainActivity::class.java)
   SalesforceSDKManager.getInstance()
     .setAllowedSPApps(listOf(
     SPConfig(
       "com....restexplorer", /* Package name of SP app */
       "com....restexplorer.ExplorerActivity", /* Main activity of SP app */
       "<-- the oauth consumer key of the sp app -->",
       "<-- the oauth callback url of the sp app -->",
       arrayOf("api", "web") /* Oauth scopes of the SP app */
   ),
```

```
 }
 /* Code here not shown */
}
```

To kick off the IDP-initiated login flow, the following example code calls `kickOffIDPInitiatedLoginFlow` for the chosen client app package name, which handles the status updates. In this example, updates are presented in a toast notification, but the application ultimately decides how to show progress to the user. You can find the corresponding version of this code in the UI of your IDP app. Examples of the selection UI can be found on GitHub, in the https://github.com/forcedotcom/SalesforceMobileSDK-Templates repo. For Android, check out the AndroidIDPTemplate.

```
SalesforceSDKManager.getInstance().idpManager?.let { idpManager ->
   idpManager.kickOffIDPInitiatedLoginFlow(this, spAppPackageName,
       object:IDPManager.StatusUpdateCallback {
           override fun onStatusUpdate(status: IDPManager.Status) {
               CoroutineScope(Dispatchers.Main).launch {
                   Toast.makeText(
                       applicationContext,
                       getString(status.resIdForDescription),
                       Toast.LENGTH_SHORT
                   ).show()
               }
           }
       }
   )
} ?: run {
   Log.e(TAG, "Cannot proceed with launch of ${appName} - not configured as IDP")
}
```

The `onStatusUpdate` callback can return any of the following status updates.

- LOGIN_REQUEST_SENT_TO_SP
- GETTING_AUTH_CODE_FROM_SERVER
- ERROR_RECEIVED_FROM_SERVER
- AUTH_CODE_SENT_TO_SP
- SP_LOGIN_COMPLETE

Configuring an Android App as an Identity Provider Client

You can configure any app built on Mobile SDK 11.0 or later as an identity provider client, as long as it's not the same app being used as an identity provider. You configure it to identify itself as an identity provider client, call a method to identify the IDP app, and Mobile SDK does the rest.

To let the client app know about the IDP app, define the IDP app package name in the `onCreate` method of its application subclass.

```
class MyApplication : Application() {

 override fun onCreate() {
   super.onCreate()
   SalesforceSDKManager.initNative(applicationContext, MainActivity::class.java)
   SalesforceSDKManager.getInstance()
     .setIDPAppPackageName(
      "com.salesforce.samples.salesforceandroididptemplateapp" /* Package name of the IDP
 app */
```

```
        )
    }
    /* Code here not shown */
}
```

That's all you need to do on the client app side. To enable logins across both the IDP and IDP client apps, make sure you've also configured the IDP app to know about the client app. Upon configuration of both apps:

- A **Login with IDP app** button appears on the login screen. When selected, this kicks off a client app-initiated login flow. The label for this button is managed by the resource string `sf__launch_idp`, which you can override within the app.
- The app now responds to IDP-initiated login requests.

Configuring an iOS App as an Identity Provider

You can configure any app built on Mobile SDK 11.0 or later as an identity provider. You configure it to identify itself as an identity provider, and Mobile SDK does the rest.

The easiest way to create an identity provider app is by using the Mobile SDK Mobile SDK iOSIDPTemplate. This template is available on GitHub in the github.com/forcedotcom/SalesforceMobileSDK-Templates repo. Use the *`forceios createwithtemplate`* command with the URI of the template repo, as shown in the following command-line example.

```
$ forceios createwithtemplate
Enter URI of repo containing template application: IOSIDPTemplate
Enter your application name: MyIDP-iOS
Enter your package name: com.acme.android
Enter your organization name (Acme, Inc.): Acme Systems
Enter output directory for your app (leave empty for the current directory): MyIDP-iOS
```

Convert an Existing Mobile SDK iOS App into an Identity Provider

To convert an existing Mobile SDK 11.x (or newer) iOS app into an identity provider:

1. In the `SalesforceSDKManager`, set `isIdentityProvider` to true.
2. In your `AppDelegate` class implementation, find the following method and reinstate the commented code as follows:

   Swift

   ```
   func application(_ app: UIApplication,
                    open url: URL,
                     options: [UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
       return UserAccountManager.shared.handleIdentityProviderCommand(
           from: url, with: options)
   }
   ```

   Objective-C

   ```
   - (BOOL)application:(UIApplication *)app
               openURL:(NSURL *)url
               options:(NSDictionary<UIApplicationOpenURLOptionsKey,id>*)options {

       return [[SFUserAccountManager sharedInstance]
           handleIDPAuthenticationResponse:url options:options];
   }
   ```

3. Add your custom URI scheme to the `info.plist` configuration. For example, the following XML defines "sampleidpapp" as a custom URI scheme:

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>sampleidpapp</string>
        </array>
    </dict>
</array>
```

To convert an existing Mobile SDK 11.x (or newer) iOS app into an identity provider, go to the `SalesforceSDKManager`. Then:

1. Set `isIdentityProvider` to `true`.
2. In `SFUserAccountManager`, initiate the flow using the following method.

```
- (void)kickOffIDPInitiatedLoginFlowForSP:(SFSDKSPConfig *)config
                            statusUpdate:(void(^)(SFSPLoginStatus))statusBlock
                                 failure:(void(^)(SFSPLoginError))failureBlock;
```

(Optional) Configure Keychain for your IDP Flow

For IDP-initiated login, you can use a shared keychain group to communicate between IDP and IDP client apps, which reduces the number of times a user has to switch between apps.

1. Add a keychain group in the "Keychain Sharing" section of your Xcode project configuration.

   If you've already configured an app group, you can use the keychain group automatically generated from the app group.

   Note: The keychain you use for IDP and the keychain you use for other Mobile SDK operations can be set independently.

2. If you configure the app under the keychain group and want to share only the IDP token without the rest of the keychain items, set `KeychainHelper.accessGroup` to the app's private keychain access group. Otherwise, the app defaults to the first keychain group in the list.

(Optional) Configure Your IDP App to Use Keychain

1. On the IDP app, go to the `SalesforceSDKManager` and set `isIdentityProvider` to `true`.
2. Initiate the flow in `SFUserAccountManager` by using this method.

```
- (void)kickOffIDPInitiatedLoginFlowForSP:(SFSDKSPConfig *)config
statusUpdate:(void(^)(SFSPLoginStatus))statusBlock
failure:(void(^)(SFSPLoginError))failureBlock;
```

(Optional) Customizing the Identity Provider UI

When a client app forwards a login request, the identity provider typically presents a selection dialog box. This dialog box, which lists known users, appears only if at least one of the following conditions is true:

- A user has logged in from any other identity provider client app before this request.

- A user has directly logged in to the identity provider app before this request.
- Multiple users are currently logged in.

Note: Note: If no users have logged in before this request, Mobile SDK displays a login screen and continues to authentication after the user successfully finishes the login flow.

To customize the user selection view, an identity provider app extends the `UIViewController` class and must also implement the `SFSDKUserSelectionView` protocol.

```
@protocol SFSDKUserSelectionViewDelegate
- (void)createNewUser:(NSDictionary *)spAppOptions;
- (void)selectedUser:(SFUserAccount *)user
        spAppContext:(NSDictionary *)spAppOptions;
- (void)cancel();

@protocol SFSDKUserSelectionView<NSObject>
    @property (nonatomic,weak) id<SFSDKUserSelectionViewDelegate> userSelectionDelegate;
    @property (nonatomic,strong) NSDictionary *spAppOptions;
@end
```

In identity provider client apps, Mobile SDK sets up an instance of the `userSelectionDelegate` and `spAppOptions` properties defined in the `SFSDKUserSelectionView` protocol. You use these objects in your identity provider's view controller to notify Mobile SDK of the user's user account selection. For example, assume that you've implemented the `SFSDKUserSelectionView` protocol in a `UIViewController` class named `UserSelectionViewController`. You can then use that view controller as the user selection dialog box by setting the `idpUserSelectionBlock` on the `SalesforceSDKManager` shared instance, as follows:

```
//optional : Customize the User Selection Screen
[SalesforceSDKManager sharedManager].idpUserSelectionBlock =
    ^UIViewController<SFSDKUserSelectionView> *{
        UserSelectionViewController *controller =
            [[UserSelectionViewController alloc] init];
        return controller;
}
```

Configuring an iOS App as an Identity Provider Client

You can configure any app built on Mobile SDK 11.0 or later as an identity provider client. You configure it to identify itself as an identity provider client and to specify its identity provider. Mobile SDK does the rest.

1. In the `init()` method of your `AppDelegate` class, specify the URI scheme for the identity provider you're using:

   Swift

   ```
   SalesforceManager.shared.identityProviderURLScheme = "sampleidpapp"
   ```

   Objective-C

   ```
   [SalesforceSDKManager sharedManager].idpAppURIScheme = @"sampleidpapp";
   ```

2. In your app's `info.plist` file, add the URI scheme defined in your identity provider clients' connected app:

   ```
   <key>CFBundleURLTypes</key>
   <array>
       <dict>
   ```

```
        <key>CFBundleURLSchemes</key>
        <array>
            <string>sampleidpclientapp</string>
        </array>
    </dict>
</array>
```

3. In your `AppDelegate` class implementation, find the following method and reinstate the commented code as follows:

   Swift

   ```
   func application(_ app: UIApplication,
                open url: URL,
                 options: [UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
       return UserAccountManager.shared.handleIdentityProviderCommand(
           from: url, with: options)
   }
   ```

   Objective-C

   ```
   - (BOOL)application:(UIApplication *)app
               openURL:(NSURL *)url
               options:(NSDictionary<UIApplicationOpenURLOptionsKey,id>*)options {

       return [[SFUserAccountManager sharedInstance]
           handleIDPAuthenticationResponse:url options:options];
   }
   ```

Your app is now ready for use as an identity provider client.

(Optional) Configure Your IDP Client App to Use Keychain

After you set up your IDP app to initiate authentication, update the client to complete the flow. You can use these methods to handle the incoming IDP URL.

Swift

```
public func handleIdentityProviderCommand(from url: URL,
with options: [AnyHashable: Any],
completion: @escaping (Result<(UserAccount, AuthInfo),
UserAccountManagerError>) -> Void) -> Bool {
    return __handleIDPAuthenticationCommand(url, options: options, completion: {
(authInfo, userAccount) in
        completion(Result.success((userAccount, authInfo)))
    }) { (authInfo, error) in
        completion(Result.failure(.loginFailed(underlyingError: error, authInfo:
authInfo)))
    }
}
```

Objective-C

```
- (BOOL)handleIDPAuthenticationCommand:(NSURL *)url
options:(nonnull NSDictionary *)options
completion:(nullable SFUserAccountManagerSuccessCallbackBlock)completionBlock
```

```
failure:(nullable SFUserAccountManagerFailureCallbackBlock)failureBlock
NS_REFINED_FOR_SWIFT;
```

(Optional) Customizing the Login Flow Selection View in the Client App

Mobile SDK provides template apps for both identity providers and their client apps. The client template defines a view that lets the user choose to log in through an identity provider or the Salesforce login screen. When a user opens an app built from the client template, the app presents this view if

- the user hasn't yet logged in, or
- the current user hasn't been set.

To customize the login style selection view, a client app extends the `UIViewController` class and also must implement the `SFSDKLoginFlowSelectionView` protocol.

```
@protocol SFSDKLoginFlowSelectionViewDelegate<NSObject>
/**
 * Used to notify the SDK of user selection on the login flow selection view
 * @param controller instance invoking this delegate
 * @param appOptions addl. name value pairs sent from the sdk for
 * the SFSDKLoginFlowSelectionView
 */
-(void)loginFlowSelectionIDPSelected:(UIViewController *)controller
    options:(NSDictionary *)appOptions;

/**
 * Used to notify the SDK of user selection on the login flow selection view
 * @param controller instance invoking this delegate
 * @param appOptions addl. name value pairs sent from the sdk for
 * the SFSDKLoginFlowSelectionView
 */
-(void)loginFlowSelectionLocalLoginSelected:(UIViewController *)controller
    options:(NSDictionary *)appOptions;

@end
```

```
@protocol SFSDKLoginFlowSelectionView<NSObject>
   @property (weak,nonatomic) id <SFSDKLoginFlowSelectionViewDelegate>selectionFlowDelegate;

    @property (nonatomic,strong) NSDictionary *appOptions;
@end
```

During the client app's identity provider flow, Mobile SDK sets up an instance of the `selectionFlowDelegate` and `appOptions` properties defined in this protocol. You use these artifacts in your view controller to notify Mobile SDK of the user's login method selection. For example, assume that you've implemented the `SFSDKUserSelectionView` protocol in a `UIViewController` class named `IDPLoginNavViewController`. You then can use that view controller as the user selection dialog box by setting the `idpLoginFlowSelectionAction` on the `SalesforceSDKManager` shared instance, as follows:

```
//optional : Customize the Login Flow Selection screen
[SalesforceSDKManager sharedManager].idpLoginFlowSelectionAction =
^UIViewController<SFSDKLoginFlowSelectionView> *{
    IDPLoginNavViewController *controller =[[IDPLoginNavViewController alloc] init];
    return controller;
}
```

Implementing Mobile Identity Provider Apps Without Mobile SDK

If you own a website that hosts apps that connect to Salesforce, you can configure Salesforce to provide identity services for those apps. Users of the hosted apps can then enjoy single sign-on ease through their host website. But can you adopt the same service in native mobile apps, and if so, what does it take? The answers are yes, and it's not difficult. You can define native mobile apps as Salesforce identity providers either of two ways: with Salesforce Mobile SDK, or without Salesforce Mobile SDK. This article gives you instructions for creating mobile identity providers and their clients without Salesforce Mobile SDK.

With identity provider (IdP) authentication flow, you can designate one trusted app as the central handler for all of a device's Salesforce login requirements. You can configure any app to be either an IdP app or an IdP client app, also know as a *service provider* (SP). Client apps delegate user authentication to your designated identity provider. The identity provider uses the customer's authenticated state to log in.

With the proper configuration, you can also implement the IdP service to non-Mobile SDK apps. Here are the requirements on the Salesforce side.

Important:

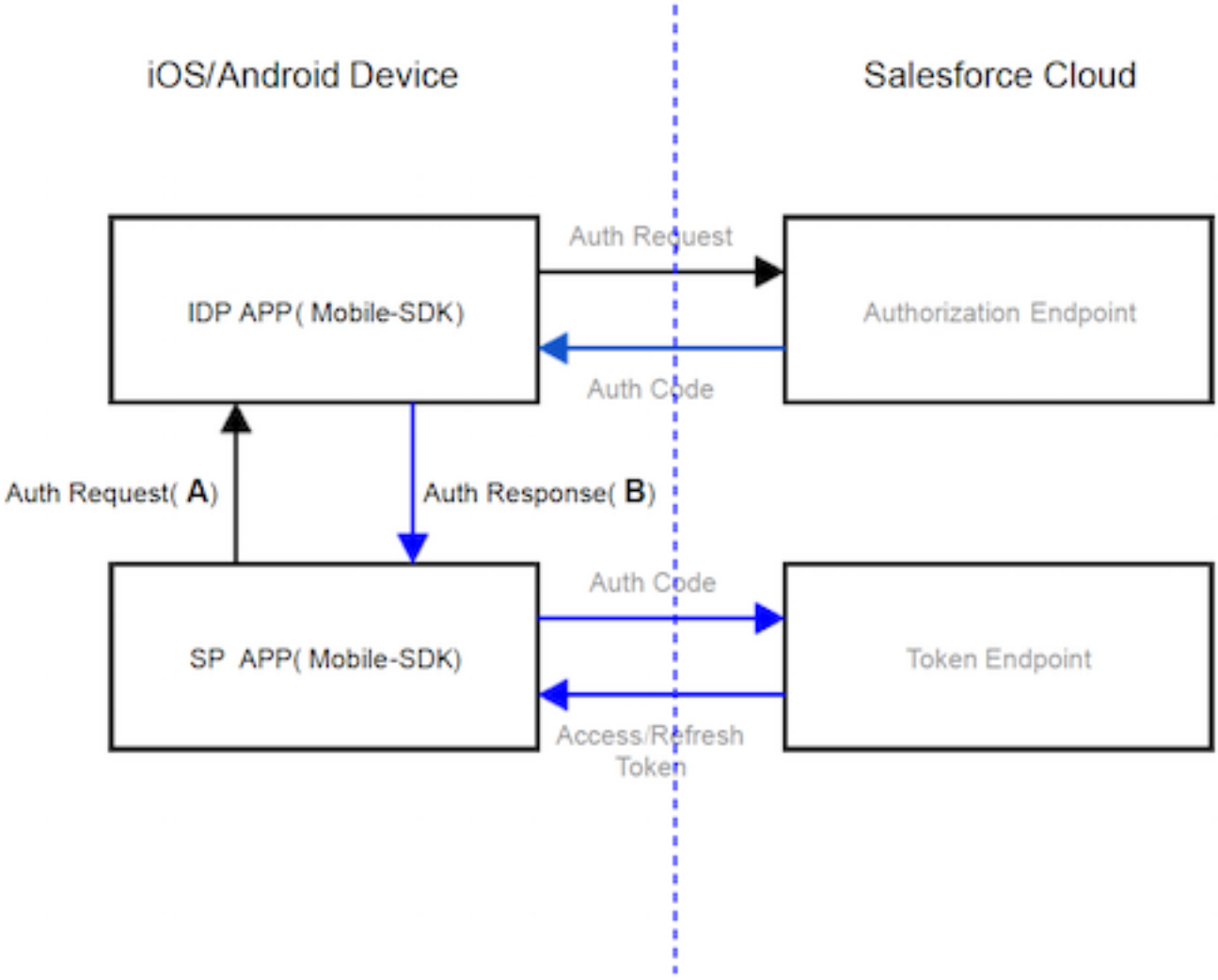
- These instructions are intended for mobile app developers who provide a suite of apps that authenticate against Salesforce but don't use Mobile SDK. For this use case, the identity provider implementation can improve the customer's experience by simulating single sign-on (SSO). However, we can provide only the description of the minimum information exchange required to centralize login authorization. **If you take this route, the mobile implementation burden lies completely with you.**
- To achieve the same result with little effort, you can use Salesforce Mobile SDK. See **Using Mobile SDK Apps as Identity Providers** in the *Salesforce Mobile SDK Development Guide*.

Rules for Identity Provider Configuration

- The identity of the IdP app is built into its client SP apps.
- The IdP app doesn't have prior knowledge of client SP apps.
- IdP apps and SP apps exchange request and response information in pre-determined formats that both apps understand. The exchange mode for this information depends on the mobile operating system.
- The IdP app maintains a list of users that are currently logged in.
- For a known user, the IdP app requests an authorization code from Salesforce. For an unrecognized user, the IdP app presents a Salesforce login screen.
- When the SP app receives an authorization code from the IdP app, the SP app directly uses the code to update the user's authentication artifacts. No tokens or passwords are shared between an SP app and its IdP.
- An IdP and its SP client apps can be published under different developer accounts.

Authentication Flow

The flow of identity provider authentication is the same regardless of whether you use Mobile SDK. Let's revisit the basic flow diagram.



For SP-initiated flows, your IdP app handles the authorization request received from any SP app (A). After negotiating with the Salesforce service, the IdP app sends the auth code received from Salesforce to the SP app (B). Your IdP app shares its custom URL scheme or package ID with its client apps.

The following sections describe the messages sent for (A) and (B).

(A) Handle an Authorization Request from an SP App

SP apps send authorization requests with the following information. IdP apps expect this information and provide code to digest it.

oauth_client_id
Client ID of SP application.

code_challenge
A cryptographic hash using the private keys of the SP app. See Proof Key for Code Exchange by OAuth Public Clients.

oauth_redirect_uri
Redirect URI for SP app. Must be registered in the SP's connected app.

scopes
Comma-separated list of requested scopes.

user_hint
For use with IDP initiated flow. Used as a loop-back param for a selected user.

login_host
Salesforce login host selected in the SP app.

state
A state value that the IdP returns to the SP app on completion of the code flow.

In iOS, the IdP app defines a custom scheme that conveys this information as parameters. The IdP app's client SP apps know this scheme. For example:

```
<IDP-APP-URL-SCHEME>://oauth2/v1.0/authrequest?
  oauth_client_id={ClientID}
  &code_challenge=base64({code challenge})
  &oauth_redirect_uri={SP APP Redirect URI}
  &scopes={requested scopes comma separated}
  &user_hint={orgid:userid}
  &login_host={loginHost}
  &state={state}
```

where <*IDP-APP-URL-SCHEME*> is the callback URL defined in the Salesforce connected app.

In Android, instead of a custom scheme, the SP app launches an intent with a bundle that contains this information as key-value pairs.

If the IdP app recognizes the user described by `user_hint`, it uses the values received to request an authorization code from the Salesforce authorization endpoint. Otherwise, it posts a Salesforce login screen and reverts to the normal Salesforce login flow.

(B) Send an Authorization Code Response from the IdP App to the Calling SP APP

When the IdP app receives an authorization code from Salesforce, it sends a response with the following information to the calling SP app.

code
Authorization code received by the IdP app as a result of the auth request (A).

login_host
Salesforce login host that was used for the auth request (A).

state
A state value that IDP App received from the SP App in the auth request (A)

In iOS, the SP app defines a custom scheme that receives these values as parameters. The SP app provides this scheme to the IdP app through the `oauth_redirect_uri` request parameter. For example:

```
<SP-APP-URL-SCHEME>://oauth2/v1.0/authresponse?
  code={authcode}
  &state={state}
  &login_host={loginHost}
```

In Android, instead of a custom scheme, the IdP app launches an intent with a bundle that contains these key-value pairs.

Error Handling

Errors use the following format.

```
<APP-URL-SCHEME>://oauth2/v1.0/error?
  error_code={error_code}
  &error_desc={description}
  &error_reason={reason}
  &state={state}
```

IN THIS SECTION:

Implementation Details and Options

Besides the basic communication details, certain standard scenarios apply to most identity provider setups.

Initiate Authentication from an IdP App

You can also design your IdP app to allow customers to use it, rather than the SP app, as the start point for authentication. In this case, your IdP app sends the following information to the SP app.

user_hint
A value providing a handle to a user reference. The SP App can use this hint to verify the customer's existence. If the user account is not found, the SP app uses this value to invoke the IdP App. The IdP app then obtains an authorization code as in (A). On the other hand, if the user account is available in the SP app, the SP app simply switches to that user.

login_host
The selected login host that was used for the auth request (A).

start_url
The URL to navigate to once the user is selected.

Use the SP app's custom scheme on iOS, or launch an intent with a bundle containing the key-value pairs on Android. For example, on iOS:

```
<SP-APP-URL-SCHEME>://oauth2/v1.0/idpinit?
  user_hint={orgid:userId}
  &login_host={loginhost}
  &start_url={starturl}
```

Fetch the Salesforce Authorization Code

Once the selected user has logged in and passed any other security requirements, your IdP app obtains an authorization code from Salesforce. How you retrieve this code depends on your identity service implementation. If the connected app policy requires permission to access the customer's data, your IdP app must also display the Salesforce permission screen.

For example, Mobile SDK launches a web view that calls a JSP server app. This server page accesses the Salesforce authorization endpoint, obtains the authorization code, and displays the access permission screen.

Verify the SP App

For protection from rogue SP apps, an IdP app can do the following.

1. Extract the calling app's bundle identifier.
2. Compare this identifier with the `redirect_uri`. If the two values don't match, return an authentication error.

By definition, the IdP app isn't expected to have prior knowledge of SP apps. However, an IdP app can choose to maintain a "white list" of apps that it intends to support.

User Selection

For user selection, the IdP app presents one of the following screens.

User selection screen—Presented if one or more users are present in the Account Manager, but none of them match the **user_hint**.

Login screen—Presented in either of the following cases:

- If no user has logged in to the IdP app.
- If the specified user has never logged in.

Return Result

Your IdP app intercepts requests that match the `oauth_redirect_uri` and calls the SP app using the provided scheme or bundle ID.

See Also

The following related links on configuring Salesforce as an identity provider are provided for general background knowledge. Most of this information is not used in configuring a mobile app as a Salesforce IdP or SP.

- Identity Providers and Service Providers in *Salesforce Help*
- Single Sign-On in *Salesforce Help*

CHAPTER 17 Using Key-Value Stores for Secure Data Storage

Beginning in Mobile SDK 8.2, encrypted key-value stores offer an alternative to SmartStore for secure data storage on mobile devices. Key-value stores aren't a replacement for SmartStore. They're designed for simpler storage scenarios that don't demand the full power of a relational database. An example is a response cache that requires your app to fetch data quickly from an opaque pool of values, unaware of data relationships or structure.

Key-value stores use AES-256 encryption and are stored on the device file system, each in its own directory. For each store, you provide a name that becomes the file name prefix. Store names can contain only letters, digits, and underscores, and can't exceed 96 characters. An app can create as many stores as the device's free space allows. Like SmartStore instances, key-value stores can be either user-based or global, depending on your use case.

Key-value stores are data-type agnostic and can contain different shapes and formats. For example, one value can be a JavaScript file, while the next is HTML, and the next a PNG image. A store doesn't recognize or require relationships between its values.

To store binary data securely, the key-value store API provides special methods. Use these methods instead of legacy techniques such as creating a JSON envelope in SmartStore, or creating a file using the Mobile SDK file encryption APIs.

For larger data sets on Android, you can buffer data and stream the values into the key-value store. For example, you can build a REST response cache and then import it by passing `restResponse.asInputStream()` to the `saveStream()` method. The key-value store reads buffers in a loop from the data source and writes them to the store file. Similar streaming isn't supported on Mobile SDK for iOS.

Key-Value Store or SmartStore?

Consider using a key-value store when:

- A simple look-up by key serves your data set well.
- Your values aren't stored as JSON or otherwise structured for atomic access.
- You're storing a large data set.
- You're storing binary data.
- You're developing a native app. In Mobile SDK 8.2, this feature isn't available for hybrid or React Native apps.

SmartStore remains a better choice if:

- Your app demands more complex querying power—for example, query predicates that filter on multiple fields, or relational queries such as joins.
- Your app must be able to retrieve parts of the data. You can't extract smaller parts of a value as you can with a SmartStore query such as `select {soup:some_indexfield} from {soup}`.
- Values are at least semi-structured. For example, your data set is JSON, but the shapes of individual values vary.
- You're developing a hybrid or React Native app.

Key-Value Store Versions

Mobile SDK 8.2, 8.3, and 9.0 use version 1 of the key-value store. Mobile SDK 9.1 introduces version 2. These versions implement the same basic functionality, but version 2 adds a public accessor that returns a list of all keys in a given store. Invoking this accessor on version 1 stores returns `nil` on iOS and throws an exception on Android.

Mobile SDK 9.1 supports both store versions but creates only version 2 stores. Because Mobile SDK knows only a one-way hash of your keys but not the keys themselves, automatic migrations aren't possible. To convert version 1 stores to version 2:

1. Create a store in Mobile SDK 9.1.
2. Use an iterative process to recreate your key-value pairs in the new store.

Key-Value Store Classes

Key-value store factory methods let you create, list, and remove stores. On iOS, these methods are part of the `KeyValueEncryptedFileStore` class. On Android, import these methods from `SmartStoreSDKManager`:

- **Android only:** `import com.salesforce.androidsdk.smartstore.app.SmartStoreSDKManager`

Store management methods for iOS and Android are in their respective `KeyValueEncryptedFileStore` classes. Use these methods to save, access, and remove values, or to count or remove key-value pairs.

- **iOS (Swift):** `KeyValueEncryptedFileStore.swift` (included in `SalesforceSDKCore`)
- **Android:** `KeyValueEncryptedFileStore.java` (`com.salesforce.androidsdk.smartstore.store` package)

Note: `KeyValueEncryptedFileStore` is a native Swift API. To access it in Objective-C, add the following line to your imports:

```
#import <SalesforceSDKCore/SalesforceSDKCore-Swift.h>
```

Set Up a Key-Value Store

1. Construct an instance of `KeyValueEncryptedFileStore`.

   iOS

   This constructor creates a store for the current user.

   ```
   let kv = KeyValueEncryptedFileStore.shared(withName: "<some name>")
   ```

   Android

   ```
   KeyValueEncryptedFileStore kv =
   SmartStoreSDKManager.getKeyValueStore("<some name>")
   ```

2. Add static key-value pairs.

   iOS (Swift)

   ```
   kv.saveValue(value, forKey: key)
   ```

   Android (Java)

   ```
   kv.saveValue(key, value)
   ```

   Or add values as input streams.

   iOS (Swift)

   Not supported

   Android (Java)

   ```
   kv.saveStream(key, stream)
   ```

Store and Retrieve Binary Data (Key-Value Store Version 2 or Later)

For managing binary data in a key-value store, Mobile SDK 10.0 introduces new iOS methods and reuses existing Android methods. Use these methods instead of the Mobile SDK file encryption APIs or a JSON envelope in SmartStore.

iOS (Swift)

```
/// Saving binary data to a key value store
/// Updates the data stored for the given key or adds a new entry
/// if the key does not exist.
/// - Parameters:
/// - data: Data to add to the store.
/// - key: Key associated with the data.
/// - Returns: True on success, false on failure.
@objc @discardableResult
public func saveData(_ data: Data, forKey key: String) → Bool

/// Accesses the data associated with the given key.
@objc public func readData(key: String) → Data?
```

Example

```
// Saving binary data to key value store
let sampleData = ...
store.saveData(sampleData, forKey:"key")

// Retrieving binary data back from key value store
let savedData = store.readData(key: "key")
```

Android (Java)

```
/**
 * Save value given as an input stream for the given key.
 * Note: This method does not close the provided input stream
 *
```

```
 * @param key Unique identifier.
 * @param stream Stream to be persisted.
 * @return True - if successful, False - otherwise.
 */
public boolean saveStream(String key, InputStream stream) throws
IOException;

* Retrieving binary data from a key value store.

/**
 * Returns stream for value of given key.
 *
 * @param key Unique identifier.
 * @return stream to value for given key or null if key not found.

 */
public InputStream getStream(String key);
```

Example

```
// Saving binary data to key value store
//

byte[] arrayToWrite = ...;

// In real life, you probably would start from a stream
// (e.g. from a network call's response)
InputStream streamToWrite =
    new ByteArrayInputStream(arrayToWrite);
keyValueStore.saveStream("key", streamToWrite);

//
// Retrieving binary data back from key value store
//

InputStream streamToRead = keyValueStore.getStream("key");
byte[] arrayRead = Encryptor.
    getByteArrayStreamFromStream(streamToRead).toByteArray();
```

Get All Keys in a Store (Key-Value Store Version 2 or Later)

These methods return all keys in the given store.

iOS (Swift)

```
/// All keys in the store
/// - Returns: all keys of stored values in a v2 store, nil if
it's a v1 store
@objc public func allKeys() -> [String]?
```

Android (Java)

```
/**
  * Get all keys.
  * NB: will throw UnsupportedOperationException for a v1 store
  */
public Set<String> keySet()
```

Get the Key-Value Store Version

These APIs let you determine a store's version at runtime.

iOS (Swift)

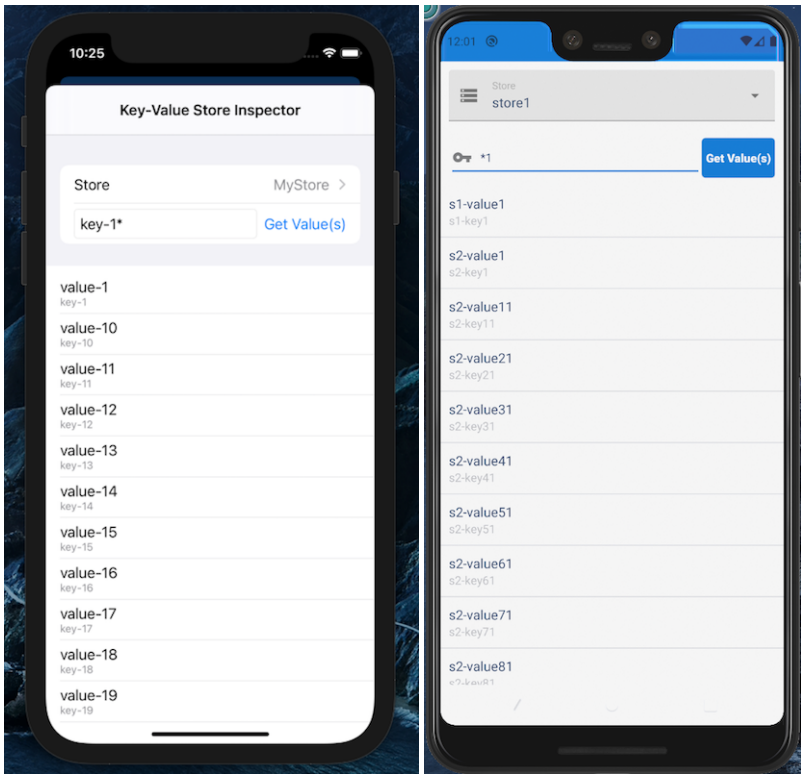
```
@objc public private(set) var storeVersion: Int
```

Android (Java)

```
public int getStoreVersion()
```

Inspect a Key-Value Store

To view a list of keys and values, select **Inspect Key-Value Store** in the Dev Support menu. This tool lets you search a store for all or part of a key name, returning all matching values.



Example: Here's an iOS Swift example:

```
...
writeToKv(value: "Joe", key: "Trader")
...

func writeToKv(value: String, key: String) {
    if let kv = KeyValueEncryptedFileStore.shared(
                    withName: "testShared") {
        if kv.saveValue(value, forKey: key) {
            let numEntries = kv.count()
            SalesforceLogger.d(RootViewController.self,
                message:"\nValue added: \(value), " +
                    "Number of entries: \(numEntries)")
        }
    }
}
```

SEE ALSO:

In-App Developer Support

CHAPTER 18 Dark Mode and Dark Theme Settings

Dark color schemes have recently become popular in user interfaces because they can reduce eye strain and improve screen readability. iOS and Android now support *dark mode* and *dark theme*, respectively, in their latest versions. To keep in sync with these developments, Mobile SDK 8.0 adds dark options for its native UI elements.

Dark schemes affect the following Mobile SDK dialog boxes:

- Choose Connection (server picker)
- Switch User
- Passcode and Biometric Input

In addition, the MobileSyncExplorer sample app and native template apps now support dark UIs.

Dark settings are managed as follows.

iOS

In Mobile SDK 8.2 and earlier, dark mode always follows the iOS setting on the mobile device. If your app isn't yet compatible with dark mode, you can disable the feature statically. To do so, open your app's `Info.plist` file and set the `UIUserInterfaceStyle` key to "Light". Apple strongly recommends that you use this setting only if you need more time to update your app's resources to dark mode. See "Choosing a Specific Interface Style for Your iOS App" at `developer.apple.com`.

Mobile SDK 8.3 introduces a `userInterfaceStyle` property on `SFSDKWindowManager` that lets you change the user interface mode at runtime. This setting applies to all windows that Mobile SDK manages as follows:

- You can choose dark, light, or unspecified mode. By default, Mobile SDK uses unspecified.
- If an app specifies the plist `UIUserInterfaceStyle` entry and also sets the `userInterfaceStyle` property, the property takes precedence.
- This property requires base SDK iOS 13 or later.

Example: These examples use iOS dark theme APIs.

```
// Swift
SFSDKWindowManager.shared().userInterfaceStyle = .dark

// Objective-C
[SFSDKWindowManager sharedManager].userInterfaceStyle =
UIUserInterfaceStyleDark;
```

Android

Dark theme in Mobile SDK apps is determined as follows:

- API 29 and higher: Defaults to the OS setting on the mobile device
- API 28 and earlier: Defaults to off
- API 23–29: Apps can force dark theme on or off

In the `SalesforceSDKManager` class, Mobile SDK provides APIs for querying and toggling dark theme. If the dark option doesn't agree with your app's existing color scheme, you can revert Mobile SDK resources to the light setting. The `Theme` enum defines the possible states:

```
public enum Theme {
    LIGHT,
    DARK,
    SYSTEM_DEFAULT
}
```

Example: These examples use Android dark theme APIs.

To query the theme setting, use this method:

```
public boolean isDarkTheme()
```

To force dark theme on or off, use the `setTheme()` method. This example turns off dark theme:

```
SalesforceSDKManager.getInstance().setTheme(SalesforceSDKManager.Theme.LIGHT);
```

CHAPTER 19 Using Experience Cloud Sites With Mobile SDK Apps

In this chapter ...

Experience Cloud sites can include up to millions of users, as allowed by Salesforce limits. With proper configuration, your customers can use their Experience Cloud site login credentials to access your Mobile SDK app. You can also brand your Experience Cloud site and login screen.

To learn more about Experience Cloud sites, see Set Up and Manage Experience Cloud Sites in Salesforce Help.

Experience Cloud Sites and Mobile SDK Apps

To enable Experience Cloud site members to log into your Mobile SDK app, set the appropriate permissions in Salesforce, and change your app's login server configuration to recognize your site URL.

With Experience Cloud sites, members that you designate can use your Mobile SDK app to access Salesforce. You define your own Experience Cloud site login endpoint, and Salesforce builds a branded site login page according to your specifications. It also lets you choose authentication providers and SAML identity providers from a list of popular choices.

Experience Cloud site membership is determined by profiles and permission sets. To enable site members to use your Mobile SDK app, configure the following:

- Make sure that each Experience Cloud site member has the API Enabled permission. You can set this permission through profiles or permission sets.
- Configure your Experience Cloud site to include your API-enabled profiles and permission sets.
- Configure your Mobile SDK app to use your Experience Cloud site's login endpoint.

In addition to these high-level steps, you must take the necessary steps to configure your users properly. Example: Configure an Experience Cloud Site For Mobile SDK App Access walks you through the Experience Cloud site configuration process for Mobile SDK apps. For the full documentation of Experience Cloud sites, see the Salesforce Help.

Note: Experience Cloud site login is supported for native and hybrid local Mobile SDK apps on iOS and Android. It is not currently supported for hybrid remote apps using Visualforce.

Set Up an API-Enabled Profile

If you're new to Experience Cloud sites, start by enabling digital experiences in your org. See Enable Digital Experiences. When you're asked to create a domain name, be sure that it doesn't use SSL (`https://`).

To set up your Experience Cloud sie, see Create an Experience Cloud Site. Note that you'll define a site URL based on the domain name you created when you enabled digital experiences.

Next, configure one or more profiles with the API Enabled permissions. You can use these profiles to enable your Mobile SDK app for Experience Cloud site members. For detailed instructions, follow the tutorial at Example: Configure an Experience Cloud Site For Mobile SDK App Access.

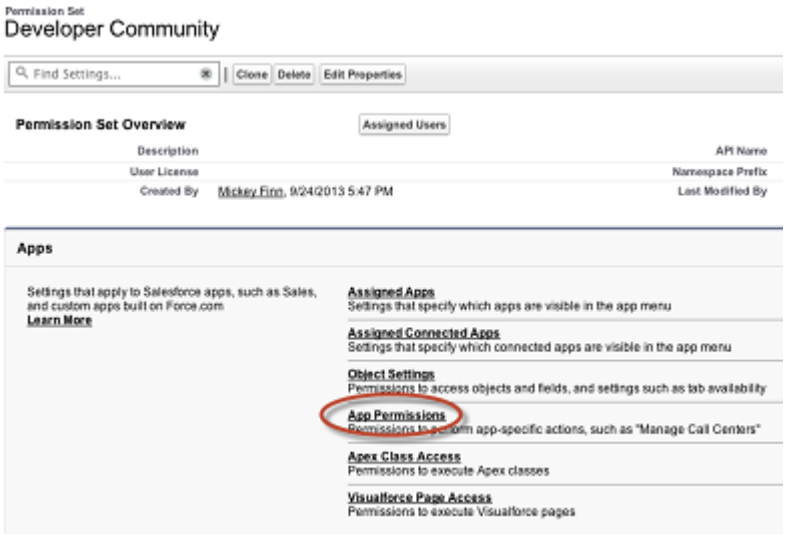
1. Create a new profile or edit an existing one.
2. Edit the profile's details to select API Enabled under **Administrative Permissions**.
3. Save your changes, and then edit your Experience Cloud site from Setup by entering *`digital experiences`* in the `Quick Find` box and then selecting **All Sites**.
4. Select **Workspaces** next to the name of your site. Then click **Administration** > **Members**.
5. Add your API-enabled profile to **Selected Profiles**.

Users to whom these profiles are assigned now have API access. For an overview of profiles, see User Profiles Overview in Salesforce Help.

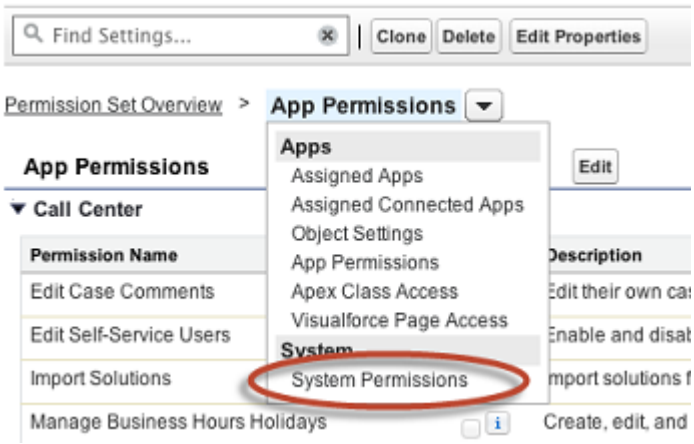
Set Up a Permission Set

Another way to enable mobile apps for your Experience Cloud site is through a permission set.

1. To add the API Enabled permission to an existing permission set, in Setup, enter *Permission Sets* in the Quick Find box, then select **Permission Sets**, select the permission set, and skip to Step 6.
2. To create a permission set, in Setup, enter *Permission Sets* in the Quick Find box, then select **Permission Sets**.
3. Click **New**.
4. Give the Permission Set a label and press *Return* to automatically create the API Name.
5. Click **Next**.
6. Under the Apps section, click **App Permissions**.



7. Click **App Permissions** and select **System** > **System Permissions**.



8. On the System Permissions page, click **Edit** and select **API Enabled**.
9. Click **Save**.
10. From Setup, enter *digital experiences* in the Quick Find box, select **All Sitees**, and click **Workspaces** next to your site name.
11. In Administration, click **Members**.

12. Under Select Permission Sets, add your API-enabled permission set to **Selected Permission Sets**.

Users in this permission set now have API access.

Grant API Access to Users

To extend API access to your Experience Cloud site users, add them to a profile or a permission set that sets the API Enabled permission. If you haven't yet configured any profiles or permission sets to include this permission, see Set Up an API-Enabled Profile and Set Up a Permission Set.

Configure the Login Endpoint

Finally, configure the app to use your Experience Cloud site login endpoint. The app's mobile platform determines how you configure this setting.

Android

In Android, login hosts are known as server connections. You can see the standard list of server connections in the `res/xml/servers.xml` file of the `SalesforceSDK` project. Mobile SDK uses this file to define production and sandbox servers.You can add your custom servers to the runtime list by creating your own `res/xml/servers.xml` file in your native Android project. The first server listed in your `servers.xml` file is used as the default login server at app startup. The root XML element for `servers.xml` is `<servers>`. This root can contain any number of `<server>` entries. Each `<server>` entry requires two attributes: `name` (an arbitrary human-friendly label) and `url` (the web address of the login server, including the "https://" prefix).

For example:

```
<?xml version="1.0" encoding="utf-8"?>
<servers>
  <server name="XYZ.com Login" url="https://myloginserver.cloudforce.com"/>
</servers>
```

iOS

Before version 4.1, Mobile SDK apps for iOS defined their custom login URIs in the app's Settings bundle. In Mobile SDK 4.1 and later, iOS apps lose the Settings bundle. Instead, you can use the `SFDCOAuthLoginHost` property in the app's `info.plist` file to build in a custom login URI.

Customers can also set their own custom login hosts at runtime in your app. Here's how:

1. Start the app without logging in.
2. In the login screen, tap the Settings, or "gear," icon  in the top navigation bar.
3. In the Choose Connection screen, tap the Plus icon  .
4. (Optional but recommended) To help identify this configuration in future visits, enter a label.
5. Enter your custom login host's URI. Be sure to omit the `https://` prefix. For example, here's how you enter a typical Experience Cloud site URI:

   ```
   MyDomainName.my.site.com/fineapps
   ```

Brand Your Experience Cloud Site

If you are using the Salesforce Tabs + Visualforce template, you can customize the look and feel of your Experience Cloud site in Experience Workspaces. You can customize by adding your company logo, colors, and copyright. Customizing these elements ensures that your Experience Cloud site matches your company's branding and is instantly recognizable to your site members.

Warning: **Mobile SDK does not support building apps that wrap Experience Builder sites.**

1. Open Experience Workspaces.
2. Click **Administration** > **Branding**.
3. Use the lookups to choose a header and footer for the Experience Cloud site.

   The files you're choosing for header and footer must have been previously uploaded to the Documents tab and must be publicly available. The header can be .html, .gif, .jpg, or .png. The footer must be an .html file. The maximum file size for .html files is 100 KB combined. The maximum file size for .gif, .jpg, or .png files is 20 KB. Let's say you have a header .html file that's 70 KB and you want to use an .html file for the footer. The footer .html file can be only 30 KB.

   The header you choose replaces the Salesforce logo below the global header. The footer you choose replaces the standard Salesforce copyright and privacy footer.
4. To select from predefined color schemes, click **Select Color Scheme**. To select a color from the color picker, click the text box next to the page section fields.

   Some of the selected colors impact your Experience Cloud site login page and how your site looks in the Salesforce mobile app as well.

Color Choice	Where it Appears
Header Background	Top of the page, under the black global header. If an HTML file is selected in the Header field, it overrides this color choice. Top of the login page. Login page in the Salesforce mobile app.
Page Background	Background color for all pages in your Experience Cloud site, including the login page.
Primary	Tab that is selected.
Secondary	Top borders of lists and tables. Button on the login page.
Tertiary	Background color for section headers on edit and detail pages.

5. Click **Save**.

Customize Login, Self-Registration, and Password Management for Your Experience Cloud Site

Configure the standard login, logout, password management, and self-registration options for your Experience Cloud site, or customize the behavior with Apex and Visualforce.

Warning: **Mobile SDK does not support building apps that wrap Experience Builder sites.**

By default, each Experience Cloud site comes with default login, password management, and self-registration pages and associated Apex controllers that drive this functionality under the hood. You can use Visualforce or Apex to create custom branding and change the default behavior. See the following steps in Salesforce Help.

- Brand your Experience Cloud site's login page.
- Customize your Experience Cloud site's login experience by modifying the default login page behavior, using a custom login page, and supporting other authentication providers.
- Redirect users to a different URL on logout.
- Use custom Change Password and Forgot Password pages.
- Set up self-registration for unlicensed guest users in your Experience Cloud site.

Use Your Branded Login Page

Starting with Mobile SDK 5.2, you can display a branded Experience Cloud site login page on your client app.

Typically, the authorization URL for a branded login page looks like this example:

```
https://MyDomainName.my.site.com/services/oauth2/authorize/<brand>?response_type=code&...
```

In this URL, `<brand>` is the branding parameter that you reuse in your app. Use the following methods to set this value, where `loginBrand` is the branding parameter for your Experience Cloud site login page.

Android

```
SalesforceSDKManager.getInstance().setLoginBrand(brandedLoginPath);
```

iOS

Swift

```
SalesforceManager.shared.brandLoginIdentifier = loginBrand
```

Objective-C

```
[SalesforceSDKManager sharedManager].brandLoginPath = loginBrand;
```

Using External Authentication With Experience Cloud Sites

You can use an external authentication provider, such as Facebook©, to log Experience Cloud site users into your Mobile SDK app.

Note: Although Salesforce supports Janrain as an authentication provider, it's primarily intended for internal use by Salesforce. We've included it here for the sake of completeness.

Authentication Provider SSO with Salesforce as the Relying Party

With authentication providers, your users can log in to your Salesforce org or Experience Cloud site with single sign-on (SSO) using credentials from a third party. Authentication providers also give your users access to protected third-party data. Salesforce offers several ways to configure authentication providers, such as with OpenID Connect or with a custom OAuth 2.0 configuration. Which protocol you can use depends on the third party.

You have several ways to configure an authentication provider.

- Predefined authentication providers
- Salesforce-managed authentication providers
- OpenID Connect authentication providers
- Custom authentication providers

After you configure an authentication provider in Salesforce, you can add it to your Salesforce login page or your Experience Cloud login page.

EDITIONS

Available in: Lightning Experience and Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

USER PERMISSIONS

To view the settings:
- View Setup and Configuration

To edit the settings:
- Customize Application

AND

Manage Auth. Providers

Single Sign-On Authentication and Authorization Flow

Most authentication providers serve a dual purpose. In addition to authenticating users for SSO, they provide access to user data. With access to this third-party data, you can enrich your users' Salesforce profiles with additional information after they log in with SSO.

For example, when a user logs in to Salesforce using their Facebook credentials, they can authorize access to their Facebook data. Facebook then sends Salesforce an access token, which you can use to access Facebook profile data in order to populate the user's Salesforce user profile.

Direct Users to an Experience Cloud Site after Authentication

When you set up single sign-on (SSO) with an authentication provider, use the Experience Cloud site URL request parameter to send users to a specific site after authenticating. With the site parameter, you can determine whether a user logs in to your Salesforce org or in to an Experience Cloud site. This parameter can also change what type of user the registration handler creates.

For example, you set up a Google authentication provider to configure SSO with your Salesforce org as the relying party. You want to send site users who log in with Google credentials to a custom site, so you add the `site` parameter to your SSO client configuration URL. A user goes to your org's login page, clicks a Google login button, and enters their Google credentials. After Google authenticates the user, Salesforce redirects the user to your custom site.

If you add the `site` parameter to the Single Sign-On Initialization URL, Salesforce sends site users to the site after they log in. If you add the parameter to the Existing User Linking URL, the **Continue to Salesforce** link on the confirmation page leads to the site.

Without the `site` parameter, Salesforce sends the user to `/home/home.jsp` for a portal or standard application, or to the default sites page for a site after authentication.

Example: When you create an authentication provider, initialization and callback URLs direct to the appropriate My Domain login or site URL. In the example, *`URLsuffix`* is the value you specified when you defined the authentication provider:

```
https://MyDomainName.my.salesforce.com/services/auth/sso/URLsuffix
```

EDITIONS

Available in: Lightning Experience and Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

USER PERMISSIONS

To view the settings:
- View Setup and Configuration

To edit the settings:
- Customize Application

AND

Manage Auth. Providers

Customize Relying Party Data Requests

When you set up single sign-on (SSO) with an authentication provider, use the scope parameter to customize data requests to a third party, like Facebook. For example, request access to the email address listed on a user's Facebook profile. You can use this parameter with every authentication provider except Janrain.

In an authentication provider SSO flow, scopes define the type of data the relying party can request. After the user logs in, the relying party sends an authorization request. The third party validates the user and sends back the access token with scopes. If the user authorizes access to the data defined by the scopes, the relying party can access the requested third-party data.

For example, you set up a Google authentication provider to configure SSO with your Salesforce org as the relying party. You want to give users the ability to view their Google Drive in your Salesforce org. So you add the scope parameter to your SSO client configuration URL and implement an `Auth.AuthToken` method to retrieve the access token with the scopes you requested. A user logs in to your org, is redirected to authenticate with Google, and then approves Salesforce to access their Google Drive. Salesforce then displays the user's Google Drive in Salesforce.

Note: Some third parties require you to pre-register scopes before you can request them.

In addition to any scopes you specify, authentication providers provide default scopes. The default scopes vary depending on the third party, but they usually limit access to basic user information. For example, the Salesforce default scope is `id`, which gives you the user's identity. To override default scopes, send scopes in a space-delimited string to the third party.

EDITIONS

Available in: Lightning Experience and Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

USER PERMISSIONS

To view the settings:
- View Setup and Configuration

To edit the settings:
- Customize Application

  AND

  Manage Auth. Providers

1. Add the `scope` parameter to a client configuration URL.
2. Use Apex `Auth.AuthToken` methods to retrieve the access token. See AuthToken Class in the Apex Reference Guide for more information.

Example: Here's an example of a `scope` parameter requesting the Salesforce scopes `api` and `web`added to the Single Sign-On Initialization URL, where:The Salesforce`api` scope allows the relying party to access Connect REST API resources, while the `web` scope allows the relying party to use the access token on the web.`https://login.salesforce.com/services/auth/sso/`***`orgID`***`/`***`URLsuffix`***`?scope=id+api+web`

- *`orgID`* is your Auth. Provider ID.
- *`URLsuffix`* is the value you specified when you defined the authentication provider.

Valid scopes vary depending on the third party, so refer to your third-party documentation. Salesforce supports these scopes.

Value	Description
Perform ANSI SQL queries on Customer Data Platform data (`cdp_query_api`)	Allows ANSI SQL queries of Data Cloud data on behalf of the user.
Manage Pardot services (`pardot_api`)	Allows access to Marketing Cloud Account Engagement API services on behalf of the user. Manage the full extent of accessible services in Account Engagement. (Pardot is now Marketing Cloud Account Engagement.)
Manage Customer Data Platform profile data (`cdp_profile_api`)	Allows access to Data Cloud REST API data. Use this scope to manage profile records.
Access Connect REST API resources (`chatter_api`)	Allows access to Connect REST API resources on behalf of the user.

Value	Description
Manage Customer Data Platform Ingestion API data (`cdp_ingest_api`)	Allows access to Data Cloud Ingestion API data. Use this scope to upload and maintain external datasets in Data Cloud. This scope is packaged in a JSON web token (JWT).
Access Analytics REST API Charts Geodata resources (`eclair_api`)	Allows access to the Analytics REST API Charts Geodata resource.
Access Analytics REST API resources (`wave_api`)	Allows access to the Analytics REST API resources.
Manage user data via APIs (`api`)	Allows access to the current, logged-in user's account using APIs, such as REST API and Bulk API 2.0. This scope also includes `chatter_api`, which allows access to Connect REST API resources.
Access custom permissions (`custom_permissions`)	Allows access to the custom permissions in an org associated with the connected app. This scope also shows whether the current user has each permission enabled.
Access the identity URL service (`id, profile, email, address, phone`)	Allows access to the identity URL service. You can request `profile`, `email`, `address`, or `phone` individually to get the same result as using `id` because they're synonymous.
Access Lightning applications (`lightning`)	Allows hybrid apps to directly obtain Lightning child sessions through the OAuth 2.0 hybrid app token flow and hybrid app refresh token flow.
Access content resources (`content`)	Allows hybrid apps to directly obtain content child sessions through the OAuth 2.0 hybrid app token flow and hybrid app refresh token flow.
Access unique user identifiers (`openid`)	Allows access to the current, logged in user's unique identifier for OpenID Connect apps. In the OAuth 2.0 user-agent flow and the OAuth 2.0 web server flow, use the `openid` scope. In addition to the access token, this scope enables you to receive a signed ID token that conforms to the OpenID Connect specifications.
Full access (`full`)	Allows access to all data accessible by the logged-in user, and encompasses all other scopes. `full` doesn't return a refresh token. You must explicitly request the `refresh_token` scope to get a refresh token.
Perform requests at any time (`refresh_token, offline_access`)	Allows a refresh token to be returned when the requesting client is eligible to receive one. With a refresh token, the app can interact with the user's data while the user is offline. This token is synonymous with requesting `offline_access`.

Value	Description
Access Visualforce applications (`visualforce`)	Allows access to customer-created Visualforce pages only. This scope doesn't allow access to standard Salesforce UIs. To allow hybrid apps to directly obtain Visualforce child sessions, include this scope with the OAuth 2.0 hybrid app token flow or hybrid app refresh token flow.
Manage user data via Web browsers (`web`)	Allows use of the `access_token` on the web. This scope also includes `visualforce`, allowing access to customer-created Visualforce pages.
Access chatbot services (`chatbot_api`)	Allows access to Einstein Bot API services.
Access Headless Registration API (user_registration_api)	Allows access to the API for the Headless Registration Flow. If you set up your flow to require authentication, you must pass in an access token that includes this scope.
Access Headless Forgot Password API (`forgot_password`)	Allows access to the API for the Headless Forgot Password Flow. If you set up your flow to require authentication, you must pass in an access token that includes this scope.
Access all Data Cloud API resources (`cdp_api`)	Allows access to all Data Cloud API resources.
Access the Salesforce API Platform (`sfap_api`)	Reserved for future use.
Access Interaction API resources (interaction_api)	Reserved for future use.

Configure a Facebook Authentication Provider

Configure a Facebook authentication provider so your users can log in to Salesforce using their Facebook credentials.

To configure Facebook as an authentication provider, complete the following steps.

1. Set up a Facebook app, making Salesforce the app domain.
2. Define a Facebook authentication provider in Salesforce.
3. Update your Facebook app to use the callback URL generated by Salesforce as the Facebook website URL.
4. Test the connection.
5. Add the Facebook provider to your login page.

EDITIONS

Available in: Lightning Experience and Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

USER PERMISSIONS

To view the settings:
- View Setup and Configuration

To edit the settings:
- Customize Application

AND

Manage Auth. Providers

Set Up a Facebook App

Before you can configure Facebook for Salesforce, you must set up an app in Facebook.

Note: You can skip this step by allowing Salesforce to use its own default app. For more information, see Use Salesforce Managed Authentication Providers.

1. Go to the Facebook website and create an app.
2. Modify the app settings, and set the Application Domain to Salesforce.

3. Note the app ID and the app secret.

Define a Facebook Provider in Salesforce

To set up a Facebook provider, you need the Facebook app ID and app secret.

Note the generated Auth. Provider ID value. You use it with the `Auth.AuthToken` Apex class.

Note: You can skip this step by allowing Salesforce to manage the values for you. For more information, see Use Salesforce Managed Authentication Providers.

1. From Setup, in the Quick Find box, enter *`Auth. Providers`*, and then select **Auth. Providers** > **New**.
2. For the provider type, select **Facebook**.
3. Enter a name for the provider.
4. Enter the URL suffix, which is used in the client configuration URLs. For example, if the URL suffix of your provider is MyFacebookProvider, your single sign-on (SSO) URL is similar to `https://`*`mydomain_url or site_url`*`/services/auth/sso/MyFacebookProvider`.
5. For Consumer Key, use the Facebook app ID.
6. For Consumer Secret, use the Facebook app secret.
7. Optionally, set the following fields and save your work.

- For Authorize Endpoint URL, enter the base URL from Facebook. For example, `https://www.facebook.com/v2.2/dialog/oauth`. If you leave this field blank, Salesforce uses the version of the Facebook API that your app uses.

  Tip: You can add query string parameters to the base URL, if necessary. For example, to get a refresh token from Facebook for offline access, use `https://accounts.facebook.com/o/oauth2/auth?access_type=offline&approval_prompt=force`. You need the `approval_prompt` parameter to ask the user to accept the refresh action so that Facebook continues to provide refresh tokens after the first one.

- For Token Endpoint URL, enter the URL from Facebook. For example, `https://www.facebook.com/v2.2/dialog/oauth`. If you leave this field blank, Salesforce uses the version of the Facebook API that your app uses.
- To change the values requested from Facebook's profile API, enter the User Info Endpoint URL. For more information, see https://developers.facebook.com/docs/facebook-login/permissions/v2.0#reference-public_profile. The requested fields must correspond to the requested scopes. If you leave this field blank, Salesforce uses the version of the Facebook API that your app uses.
- To automatically enable the OAuth 2.0 Proof Key for Code Exchange (PKCE) extension, which improves security, select **Use Proof Key for Code Exchange (PKCE) Extension**. For more information on how this setting helps secure your provider, see Build the PKCE Extension into Your Implementations.
- For Default Scopes, enter the scopes to send along with the request to the authorization endpoint. Otherwise, the hard-coded defaults for the provider type are used. See Facebook's developer documentation for these defaults.

  For more information, see Use the Scope URL Parameter.

- If you enter a consumer key and consumer secret, the consumer secret is included in SOAP API responses by default. To hide the secret in SOAP API responses, deselect **Include Consumer Secret in SOAP API Responses**. Starting in November 2022, the secret is always replaced in Metadata API responses with a placeholder value. On deployment, replace the placeholder with your consumer secret as plain text, or modify the value later through the UI.
- For Custom Error URL, enter the URL for the provider to use to report any errors.

- For Custom Logout URL, enter a URL to provide a specific destination for users after they log out, if they authenticated using the SSO flow. Use this field to direct users to a branded logout page or destination other than the default Salesforce logout page. The URL must be fully qualified with an http or https prefix, such as `https://acme.my.salesforce.com`.

  Tip: Configure single logout (SLO) to automatically log out a user from Salesforce and the identity provider. As the relying party, Salesforce supports OpenID Connect SLO when the user logs out from the identity provider or Salesforce.

- Select an existing Apex class as the `Registration Handler` class. Or to create an Apex class template for the registration handler, click **Automatically create a registration handler template**. Edit this class later, and modify the default content before using it.

  Note: A `Registration Handler` class is required for Salesforce to generate the SSO initialization URL.

- For Execute Registration As, select the user that runs the Apex handler class.

  Execute Registration As provides the context in which the registration handler runs. Select a user regardless of whether you're specifying an existing registration handler class or creating one from the template. In production, you typically create a system user for the Execute Registration As user. This way, operations performed by the handler are easily traced back to the registration process. For example, if a contact is created, the system user creates it.

- To use a portal with your provider, select the portal from the Portal dropdown list.
- For Icon URL, add a path to an icon to display as a button on the login page for a site. This icon applies to an Experience Cloud site only. It doesn't appear on your Salesforce login page or My Domain login URL. Users click the button to log in with the associated authentication provider for the site.

  Specify a path to your own image, or copy the URL for one of our sample icons into the field.

- To use the Salesforce multi-factor authentication (MFA) functionality instead of your identity provider's MFA service, select **Use Salesforce MFA for this SSO provider**. This setting triggers MFA only for users who have MFA applied to them directly. For more information, see Use Salesforce MFA for SSO.

Several client configuration URLs are generated after defining the authentication provider.

- Test-Only Initialization URL—Salesforce admins use this URL to ensure that the third-party provider is set up correctly. The admin opens this URL in a browser, signs in to the third party, and is redirected to Salesforce with a map of attributes.
- Single Sign-On Initialization URL—Use this URL to perform SSO into Salesforce from a third party using its third-party credentials. The user opens this URL in a browser and logs in to the third party. The third party creates a user or updates an existing user. Then the third party signs the user into Salesforce as that user.
- Existing User Linking URL—Use this URL to link existing Salesforce users to a third-party account. The user opens this URL in a browser, signs in to the third party, signs in to Salesforce, and approves the link.
- OAuth-Only Initialization URL—Use this URL to obtain OAuth access tokens for a third party. Users must authenticate with Salesforce for the third-party service to get a token. This flow doesn't provide for future SSO functionality.
- Callback URL—Use this URL for the endpoint that the authentication provider calls back to for configuration. The authentication provider must redirect to the callback URL with information for each client configuration URL.

Client configuration URLs support additional request parameters that enable you to direct users to log in to specific sites, obtain customized permissions from the third party, or go to a specific location after authenticating.

Update Your Facebook App

After defining the Facebook authentication provider in Salesforce, go back to Facebook and update your app to use the callback URL as the Facebook Website Site URL.

Test the SSO Connection

In a browser, open the Test-Only Initialization URL on the Auth. Provider detail page. It redirects you to Facebook and asks you to sign in. You're then asked to authorize your app. After you authorize, you're redirected to Salesforce.

Add the Authentication Provider to Your Login Page

Configure your login page to show the authentication provider as a login option. Depending on whether you're configuring SSO for an org or Experience Cloud site, this step is different.

- For orgs, see Add an Authentication Provider to Your Org's Login Page.
- For Experience Cloud sites, see Add an Authentication Provider to Your Experience Cloud Site's Login Page.

Configure a Salesforce Authentication Provider

Configure a Salesforce authentication provider so your users can log in to your custom external web app using their Salesforce credentials.

Configuring a Salesforce authentication provider involves these high-level steps.

1. Define the Salesforce authentication provider in your org.
2. Test the connection.
3. Add the authentication provider to your login page.

EDITIONS

Available in: Lightning Experience and Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

USER PERMISSIONS

To view the settings:
- View Setup and Configuration

To edit the settings:
- Customize Application

  AND

  Manage Auth. Providers

Define the Salesforce Authentication Provider in Your Org

To set up the authentication provider in your org, you need the values from the Consumer Key and Consumer Secret fields of the connected app definition.

Note: You can skip this step by allowing Salesforce to manage the values for you. For more information, see Use Salesforce Managed Authentication Providers.

1. From Setup, enter *Auth. Providers* in the Quick Find box, and then select **Auth. Providers** > **New**.
2. For the provider type, select **Salesforce**.
3. Enter a name for the provider.
4. Enter the URL suffix, which is used in the client configuration URLs. For example, if the URL suffix of your provider is MySFDCProvider, your SSO URL is similar to `https://mydomain_url or site_url /services/auth/sso/MySFDCProvider`.
5. Paste the consumer key value from the connected app definition into the Consumer Key field.
6. Paste the consumer secret value from the connected app definition into the Consumer Secret field.
7. Optionally, set the following fields.
   - For Authorize Endpoint URL, specify an OAuth authorization URL.

     For Authorize Endpoint URL, the host name can include a sandbox or company-specific custom domain login URL. The URL must end in `.salesforce.com`, and the path must end in `/services/oauth2/authorize`. For example, `https://login.salesforce.com/services/oauth2/authorize`.
   - For Token Endpoint URL, specify an OAuth token URL.

For Token Endpoint URL, the host name can include a sandbox or custom domain name. The URL must end in `.salesforce.com`, and the path must end in `/services/oauth2/token`. For example, `https://login.salesforce.com/services/oauth2/token`.

- To automatically enable the OAuth 2.0 Proof Key for Code Exchange (PKCE) extension, which improves security, select **Use Proof Key for Code Exchange (PKCE) Extension**. For more information on how this setting helps secure your provider, see Build the PKCE Extension into Your Implementations.
- For Default Scopes, enter the scopes to send along with the request to the authorization endpoint. Otherwise, the hard-coded default is used.

  For more information, see Customize Relying Party Data Requests.
- If the authentication provider was created after the Winter '15 release, the **Include identity organization's organization ID for third-party account linkage** option no longer appears. Before Winter '15, the destination org couldn't differentiate between users with the same user ID on different orgs. For example, two users with the same user ID in different orgs were seen as the same user. As of Winter '15, user identities contain the org ID, so this option isn't needed. For older authentication providers, enable this option to keep identities separate in the destination org. When you enable this option, your users must reapprove all third-party links. The links are listed in the Third-Party Account Links section of a user's detail page.
- If you enter a consumer key and consumer secret, the consumer secret is included in SOAP API responses by default. To hide the secret in SOAP API responses, deselect **Include Consumer Secret in SOAP API Responses**. Starting in November 2022, the secret is always replaced in Metadata API responses with a placeholder value. On deployment, replace the placeholder with your consumer secret as plain text, or modify the value later through the UI.
- For Custom Error URL, enter the URL for the provider to use to report any errors.
- For Custom Logout URL, enter a URL to provide a specific destination for users after they log out, if they authenticated using the SSO flow. Use this field to direct users to a branded logout page or destination other than the default Salesforce logout page. The URL must be fully qualified with an http or https prefix, such as `https://acme.my.salesforce.com`.

Tip: Configure single logout (SLO) to automatically log out a user from Salesforce and the identity provider. As the relying party, Salesforce supports OpenID Connect SLO when the user logs out from the identity provider or Salesforce.

8. Select an existing Apex class as the `Registration Handler` class. Or select **Automatically create a registration handler template** to create an Apex class template for the registration handler. Edit this class later, and modify the default content before using it.

   Note: A `Registration Handler` class is required for Salesforce to generate the SSO initialization URL.

9. For Execute Registration As, select the user that runs the Apex handler class. The user must have the Manage Users permission. A user is required regardless of whether you're specifying an existing registration handler class or creating one from the template.
10. To use a portal with your provider, select the portal from the Portal dropdown list.
11. For Icon URL, add a path to an icon to display as a button on the login page for a site. This icon applies to an Experience Cloud site only. It doesn't appear on your Salesforce login page or My Domain login URL. Users click the button to log in with the associated authentication provider for the site.

    Specify a path to your own image, or copy the URL for one of our sample icons into the field.
12. To use the Salesforce multi-factor authentication (MFA) functionality instead of your identity provider's MFA service, select **Use Salesforce MFA for this SSO provider**. This setting triggers MFA only for users who have MFA applied to them directly. For more information, see Use Salesforce MFA for SSO.
13. Click **Save**.

    Note the value of the Client Configuration URLs. You need the callback URL to complete the last step. Use the Test-Only initialization URL to check your configuration. Also note the Auth. Provider ID value because you use it with the `Auth.AuthToken` Apex class.

14. Return to the connected app definition that you created earlier from Setup. Paste the callback URL value from the authentication provider into the Callback URL field.

Several client configuration URLs are generated after defining the authentication provider.

Client configuration URLs support additional request parameters that enable you to direct users to log in to specific sites, obtain customized permissions from the third party, or go to a specific location after authenticating.

Test the SSO Connection

In a browser, open the Test-Only Initialization URL on the Auth. Provider detail page. It redirects you to the authentication provider and asks you to sign in. You're then asked to authorize your app. After you authorize, you're redirected to Salesforce.

Add the Authentication Provider to Your Login Page

- For orgs, see Add an Authentication Provider to Your Org's Login Page.
- For Experience Cloud sites, see Add an Authentication Provider to Your Experience Cloud Site's Login Page.

Configure an Authentication Provider Using OpenID Connect

To configure single sign-on (SSO) with Salesforce as the relying party for a third-party OpenID provider, set up an authentication provider that implements OpenID Connect. With this configuration, your users can log in to Salesforce from the OpenID provider and authorize Salesforce to access protected data.

You can configure an authentication provider for any third party that implements the server side of the OpenID Connect protocol. Here are some common OpenID providers.

- Amazon
- Google
- PayPal

To configure Salesforce as the relying party for your OpenID provider, complete these steps.

- Register your app, making Salesforce the app domain.
- Define an OpenID Connect authentication provider in Salesforce.
- Update your app to use the callback URL generated by Salesforce.
- Test the connection.
- Add the authentication provider to your login page.

EDITIONS

Available in: Lightning Experience and Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

USER PERMISSIONS

To view the settings:
- View Setup and Configuration

To edit the settings:
- Customize Application

AND

Manage Auth. Providers

Register an App in the OpenID Provider

Before you can define your authentication provider in Salesforce, you must register a web app with your OpenID provider. The process varies depending on the OpenID provider. For example, to register a Google app, Create an OAuth 2.0 Client ID.

1. Register your app on your OpenID provider's website.
2. Modify the app settings and set the app domain, or Home Page URL, to Salesforce.
3. From the OpenID provider's documentation, get these configuration values:
   - Client ID
   - Client Secret

- Authorization Endpoint URL
- Token Endpoint URL
- User Info Endpoint URL

Define an Authentication Provider in Salesforce

Be sure to note the generated Auth. Provider ID value. Use it with the `Auth.AuthToken` Apex class.

1. From Setup, in the Quick Find box, enter *`Auth`*, and then select **Auth. Providers**.
2. Click **New**.
3. For the provider type, select **OpenID Connect**.
4. Enter a name for the provider.
5. Enter the URL suffix, which is used in the client configuration URLs. For example, if the URL suffix of your authentication provider is MyOpenIDConnectProvider, your SSO URL is similar to `https://`*`mydomain_url or site_url`*`/services/auth/sso/OpenIDConnectProvider`.
6. For Consumer Key, use the client ID from your OpenID provider.
7. For Consumer Secret, use the client secret from your OpenID provider.
8. For Authorize Endpoint URL, enter the base URL from your OpenID provider.

   Tip: You can add query string parameters to the base URL, if necessary. For example, to get a refresh token from Google for offline access, use `https://accounts.google.com/o/oauth2/auth?access_type=offline&approval_prompt=force`. You need the `approval_prompt` parameter to ask the user to accept the refresh action so that Google continues to provide refresh tokens after the first one.

9. Enter the token endpoint URL from your OpenID provider.
10. Enter the User Info Endpoint URL from your OpenID provider.

    Note: If you want to integrate the provider with your API using OAuth, a User Info endpoint isn't required.

11. To automatically enable the OAuth 2.0 Proof Key for Code Exchange (PKCE) extension, which improves security, select **Use Proof Key for Code Exchange (PKCE) Extension**. For more information on how this setting helps secure your provider, see Build the PKCE Extension into Your Implementations .
12. Optionally, set these fields.
    - The Token Issuer field identifies the source of the authentication token in the form `https://URL`.

      For an OAuth 2.0 web server authentication flow, the provider must include an ID token in the response from the token endpoint. Optionally, the provider can include an ID token in the response for a refresh token flow.

      The ID token is validated against the Token Issuer value and information in the UserInfo endpoint. The signature of the ID token isn't validated.

      The audience for the ID token is the consumer key registered with your authentication provider. Don't include any other audience values.

    - For Default Scopes, enter the scopes to send along with the request to the authorization endpoint. Otherwise, the hard-coded defaults for the authentication provider type are used. See the OpenID Connect developer documentation for these defaults.

      For Default Scopes, enter the scopes to send along with the request to the authorization endpoint. Otherwise, the hard-coded defaults for the authentication provider type are used. See the OpenID Connect developer documentation for these defaults.

- If you enter a consumer key and consumer secret, the consumer secret is included in SOAP API responses by default. To hide the secret in SOAP API responses, deselect **Include Consumer Secret in SOAP API Responses**. Starting in November 2022, the secret is always replaced in Metadata API responses with a placeholder value. On deployment, replace the placeholder with your consumer secret as plain text, or modify the value later through the UI.

13. Optionally, to have the token sent in a header instead of a query string, select **Send access token in header**.
14. Optionally, to send the consumer key and secret in a Basic header instead of in the request body, select **Send client credentials in header**.
15. Optionally, set these fields.
    - For Custom Error URL, enter the URL for the provider to use to report any errors.
    - For Custom Logout URL, enter a URL to provide a specific destination for users after they log out, if they authenticated using the SSO flow. Use this field to direct users to a branded logout page or destination other than the default Salesforce logout page. The URL must be fully qualified with an http or https prefix, such as `https://acme.my.salesforce.com`.

      Tip: Configure single logout (SLO) to automatically log out a user from Salesforce and the identity provider. As the relying party, Salesforce supports OpenID Connect SLO when the user logs out from the identity provider or Salesforce.
    - Select an existing Apex class as the `Registration Handler` class. Or to create an Apex class template for the registration handler, click **Automatically create a registration handler template**. Edit this class later, and modify the default content before using it.

      Note: A `Registration Handler` class is required for Salesforce to generate the SSO initialization URL.
    - For Execute Registration As, select the user that runs the Apex handler class. The user must have the Manage Users permission.

      Execute Registration As provides the context in which the registration handler runs. Select a user regardless of whether you're specifying an existing registration handler class or creating one from the template. In production, you typically create a system user for the Execute Registration As user. This way, operations performed by the handler are easily traced back to the registration process. For example, if a contact is created, the system user creates it.
    - To use a portal with your provider, select the portal from the Portal dropdown list.
    - For Icon URL, add a path to an icon to display as a button on the login page for a site. This icon applies to an Experience Cloud site only. It doesn't appear on your Salesforce login page or My Domain login URL. Users click the button to log in with the associated authentication provider for the site.

      Specify a path to your own image, or copy the URL for one of our sample icons into the field.
16. To use the Salesforce multi-factor authentication (MFA) functionality instead of your identity provider's MFA service, select **Use Salesforce MFA for this SSO provider**. This setting triggers MFA only for users who have MFA applied to them directly. For more information, see Use Salesforce MFA for SSO.
17. Save the settings.

Several client configuration URLs are generated after defining the authentication provider.

Client configuration URLs support additional request parameters that enable you to direct users to log in to specific sites, obtain customized permissions from the third party, or go to a specific location after authenticating.

Update Your OpenID Connect App

After defining the authentication provider in Salesforce, go back to your OpenID provider and update your app's callback URL. For Google apps, the callback URL is called the Authorized Redirect URI. For PayPal, it's called the Return URL.

Test the SSO Connection

In a browser, open the Test-Only Initialization URL on the Auth. Provider Setup page. It redirects you to your OpenID provider and asks you to sign in. You're then asked to authorize your app. After you authorize, you're redirected to Salesforce.

Add the Authentication Provider to Your Login Page

Configure your login page to show the authentication provider as a login option. Depending on whether you're configuring SSO for an org or Experience Cloud site, this step is different.

- For orgs, see Add an Authentication Provider to Your Org's Login Page.
- For Experience Cloud sites, see Add an Authentication Provider to Your Experience Cloud Site's Login Page.

Example: Configure an Experience Cloud Site For Mobile SDK App Access

Configuring your Experience Cloud site to support logins from Mobile SDK apps can be tricky. This tutorial helps you see the details and correct sequence first-hand.

When you configure Experience Cloud site users for mobile access, sequence and protocol affect your success. For example, a user that's not associated with a contact cannot log in on a mobile device. Here are some important guidelines to keep in mind:

- Create users only from contacts that belong to accounts. You can't create the user first and then associate it with a contact later.
- Be sure you've assigned a role to the owner of any account you use. Otherwise, the user gets an error when trying to log in.
- When you define a custom login host in an iOS app, be sure to remove the `http[s]://` prefix. The iOS core appends the prefix at runtime. Explicitly including it could result in an invalid address.

1. Add Permissions to a Profile
2. Create an Experience Cloud Site
3. Add the API User Profile To Your Experience Cloud Site
4. Create a New Contact and User
5. Test Your New Experience Cloud Site Login

Add Permissions to a Profile

Create a profile that has API Enabled and Enable Chatter permissions.

1. From Setup, enter *`Profiles`* in the `Quick Find` box, then select **Profiles**.
2. Click **New Profile**.
3. For Existing Profile select **Customer Community User**.
4. For **Profile Name** type *`FineApps API User`*.
5. Click **Save**.
6. On the FineApps API User page, click **Edit**.
7. For **Administrative Permissions** select **API Enabled** and **Enable Chatter**.

   Note: A user who doesn't have the Enable Chatter permission gets an insufficient privileges error immediately after successfully logging into your Experience Cloud site in Salesforce.

8. Click **Save**.

Note: In this tutorial we use a profile, but you can also use a permission set that includes the required permissions.

Create an Experience Cloud Site

Create an Experience Cloud site and a site login URL.

The following steps are fully documented at Enable Digital Experiences and Create an Experience Cloud Site in Salesforce Help.

1. In Setup, enter *digital experiences* in the Quick Find box.
2. If you don't see **All Sites**:
   a. Click **Settings**.
   b. Select **Enable digital experiences**.
   c. Enter a unique name for your domain name, such as *fineapps.<your_name>.force.com* for **Domain name**.
   d. Click **Check Availability** to make sure the domain name isn't already being used.
   e. Click **Save**.
3. From Setup, enter *digital experiences* in the Quick Find box, then select **All Sites**.
4. Click **New Site**.
5. Choose a template and name the new Experience Cloud site *FineApps Users*.
6. For **URL**, type *customers* in the suffix edit box.
   The full URL shown, including your suffix, becomes the new URL for your Experience Cloud site.
7. Click **Create Site**.

Add the API User Profile To Your Experience Cloud Site

Add the API User profile to your Experience Cloud site setup on the Members page.

1. Click **Administration** > **Members**.
2. For Search, select **All**.
3. Select **FineApps API User** in the Available Profiles list and then click **Add**.
4. Click **Save**.
5. Click **Publish**.
6. Dismiss the confirmation dialog box and click **Close**.

Create a New Contact and User

Instead of creating users directly, create a contact on an account and then create the user from that contact.

If you don't currently have any accounts,

1. Click the **Accounts** tab.
2. If your org doesn't yet contain any accounts:
   a. In Quick Create, enter *My Test Account* for **Account Name**.

   b. Click **Save**

3. In Recent Accounts click **My Test Account** or any other account name. Note the Account Owner's name.
4. From Setup, enter *Users* in the Quick Find box, select **Users**, and then click **Edit** next to your Account Owner's name.
5. Make sure that **Role** is set to a management role, such as CEO.
6. Click **Save**.
7. Click the **Accounts** tab and again click the account's name.
8. In Contacts, click **New Contact**.
9. Fill in the following information: First Name: *Jim*, Last Name: *Parker*. Click **Save**.
10. On the Contact page for Jim Parker, click **Manage External User** and then select **Enable Customer User**.
11. For User License select **Customer Community**.
12. For Profile select the FineApps API User.
13. Use the following values for the other required fields:

Field	Value
Email	Enter your active valid email address.
Username	*jimparker@fineapps.com*
Nickname	*jimmyp*

    You can remove any non-required information if it's automatically filled in by the browser.

14. Click **Save**.
15. Wait for an email to arrive in your inbox welcoming Jim Parker and then click the link in the email to create a password. Set the password to "mobile333".

Test Your New Experience Cloud Site Login

Test your Experience Cloud site setup by logging in to your Mobile SDK native or hybrid local app as your new contact.

To log in to your Experience Cloud site from your Mobile SDK app, configure your app to recognize your site login URL.

1. For Android:
   a. Open your Android project in Android Studio.
   b. In the Project Explorer, go to the `res` folder and create a new (or select the existing) `xml` folder.
   c. In the `xml` folder, create a text file. You can do this using either the **File** menu or the *CTRL-Click* (or *Right-Click*) menu.
   d. In the new text file, add the following XML. Replace the server URL with your Experience Cloud site login URL:

```
<?xml version="1.0" encoding="utf-8"?>
<servers>
 <server name="Experience Cloud Site Login" url=
 "https://fineapps-dev-ed.my.site.com/fineapps">
</servers>
```

e. Save the file as `servers.xml`.

2. For iOS:
   a. Start the app without logging in.
   b. In the login screen, tap the Settings, or "gear," icon in the top navigation bar.
   c. In the Choose Connection screen, tap the Plus icon .
   d. (Optional but recommended) To help identify this configuration in future visits, enter a label.
   e. Enter your custom login host's URI. Be sure to omit the `https://` prefix. For example, here's how you enter a typical Experience Cloud site URI:

      ```
      MyDomainName.my.site.com/fineapps
      ```

   Alternatively, set the login screen through MDM if you're using MDM for configuration.

3. Start your app on your device, simulator, or emulator, and log in with username *jimparker@fineapps.com* and password *mobiletest1234*.

Note: If your mobile app remains at the login screen for an extended time, you can get an "insufficient privileges" error upon login. In this case, close and reopen the app, and then log in immediately.

Example: Configure an Experience Cloud Site For Facebook Authentication

You can extend the reach of your Experience Cloud site by configuring an external authentication provider to handle site logins.

This example extends the previous example to use Facebook as an authentication front end. In this simple scenario, we configure the external authentication provider to accept any authenticated Facebook user into the Experience Cloud site.

If your Experience Cloud site is already configured for mobile app logins, you don't need to change your mobile app or your connected app to use external authentication. Instead, you define a Facebook app, a Salesforce Auth. Provider, and an Auth. Provider Apex class. You also make a minor change to your Experience Cloud site setup.

Create a Facebook App

To enable Experience Cloud site logins through Facebook, start by creating a Facebook app.

A Facebook app is comparable to a Salesforce connected app. It is a container for settings that govern the connectivity and authentication of your app on mobile devices.

1. Go to developers.facebook.com.
2. Log in with your Facebook developer account, or register if you're not a registered Facebook developer.
3. Go to **Apps** > **Create a New App**.
4. Set display name to "FineApps Experience Cloud Site Test".
5. Add a Namespace, if you want. Per Facebook's requirements, a namespace label must be twenty characters or less, using only lowercase letters, dashes, and underscores. For example, "my_fb_goodapps".
6. For Category, choose **Utilities**.
7. Copy and store your App ID and App Secret for later use.

You can log in to the app using the following URL:

`https://developers.facebook.com/apps/<App ID>/dashboard/`

Define a Salesforce Auth. Provider

To enable external authentication in Salesforce, create an Auth. Provider.

External authentication through Facebook requires the App ID and App Secret from the Facebook app that you created in the previous step.

1. In Setup, enter *Auth. Providers* in the Quick Find box, then select **Auth. Providers**.
2. Click **New**.
3. Configure the Auth. Provider fields as shown in the following table.

Field	Value
Provider Type	Select **Facebook**.
Name	Enter *FB Community Login*.
URL Suffix	Accept the default. Note: You may also provide any other string that conforms to URL syntax, but for this example the default works best.
Consumer Key	Enter the App ID from your Facebook app.
Consumer Secret	Enter the App Secret from your Facebook app.
Custom Error URL	Leave blank.

4. For Registration Handler, click **Automatically create a registration handler template**.
5. For Execute Registration As:, click Search and choose an Experience Cloud site member who has administrative privileges.
6. Leave Portal blank.
7. Click **Save**.
   Salesforce creates a new Apex class that extends `RegistrationHandler`. The class name takes the form *AutocreatedRegHandlerxxxxxx*....
8. Copy the Auth. Provider ID for later use.
9. In the detail page for your new Auth. Provider, under Client Configuration, copy the Callback URL for later use.
   The callback URL takes the form
   `https://login.salesforce.com/services/authcallback/<id>/<Auth.Provider_URL_Suffix>`.

Configure Your Facebook App

Next, you need to configure the Experience Cloud site to use your Salesforce Auth. Provider for logins.

Now that you've defined a Salesforce Auth. Provider, complete the authentication protocol by linking your Facebook app to your Auth. Provider. You provide the Salesforce login URL and the callback URL, which contains your Auth. Provider ID and the Auth. Provider's URL suffix.

1. In your Facebook app, go to **Settings**.
2. In App Domains, enter *MyDomainName.my.salesforce.com*.
3. Click **+Add Platform**.
4. Select **Website**.
5. For Site URL, enter your Auth. Provider's callback URL.
6. For **Contact Email**, enter your valid email address.
7. In the left panel, set Status & Review to **Yes**. With this setting, all Facebook users can use their Facebook logins to create user accounts in your Experience Cloud site.
8. Click **Save Changes**.
9. Click **Confirm**.

Customize the Auth. Provider Apex Class

Use the Apex class for your Auth. Provider to define filtering logic that controls who may enter your Experience Cloud site.

1. In Setup, enter *Apex Classes* in the `Quick Find` box, then select **Apex Classes**.
2. Click **Edit** next to your Auth. Provider class. The default class name starts with "AutocreatedRegHandlerxxxxxx…"
3. To implement the `canCreateUser()` method, simply return true.

```
global boolean canCreateUser(Auth.UserData data) {
 return true;
}
```

This implementation allows anyone who logs in through Facebook to join your Experience Cloud site.

Note: If you want your Experience Cloud site to be accessible only to existing members, implement a filter to recognize every valid user. Base your filter on any unique data in the Facebook packet, such as username or email address, and then validate that data against similar fields in your Experience Cloud site members' records.

4. Change the `createUser()` code:
   a. Replace "Acme" with *FineApps* in the account name query.
   b. Replace the username suffix ("@acmecorp.com") with *@fineapps.com*.
   c. Change the profile name in the profile query ("Customer Portal User") to *API Enabled*.
5. In the `updateUser()` code, replace the suffix to the username ("myorg.com") with *@fineapps.com*.
6. Click **Save**.

Configure Your Experience Cloud Site

For the final step, configure the Experience Cloud site to use your Salesforce Auth. Provider for logins.

1. In Setup, enter *digital experiences* in the `Quick Find` box, then select **All Sites**.
2. Click **Manage** next to your site name.

3. Click **Administration** > **Login & Registration**.
4. Under Login, select your new Auth. Provider.
5. Click **Save**.

You're done! Now, when you log into your mobile app using your Experience Cloud site login URL, look for an additional button inviting you to log in using Facebook. Click the button and follow the on-screen instructions to see how the login works.

To test the external authentication setup in a browser, customize the Single Sign-On Initialization URL (from your Auth. Provider) as follows:

```
https://MyDomainName.my.salesforce.com/services/auth/sso/orgID/
URLsuffix?community=ExperienceCloudSite_login_url
```

For example:

```
https://MyDomainName.my.salesforce.com/services/auth/sso/00Da0000000TPNEAA4/
FB_Community_Login?community=
https://MyDomainName.my.site.com/fineapps
```

To form the Existing User Linking URL, replace `sso` with `link`:

```
https://MyDomainName.my.salesforce.com/services/auth/link/00Da0000000ABCDEF9/
FB_Community_Login?community=
https://MyDomainName.my.site.com/fineapps
```

CHAPTER 20 Multi-User Support in Mobile SDK

In this chapter ...

If you need to enable simultaneous logins for multiple users, Mobile SDK provides a basic implementation and APIs for user switching.

Mobile SDK provides a default dialog box that lets the user select from authenticated accounts. Your app implements some means of launching the dialog box and calls the APIs that initiate the user switching workflow.

About Multi-User Support

Beginning in version 2.2, Mobile SDK supports simultaneous logins from multiple user accounts. These accounts can represent different users from the same organization, or different users on different organizations (such as production and sandbox, for instance.)

Once a user signs in, that user's credentials are saved to allow seamless switching between accounts, without the need to re-authenticate against the server. If you don't wish to support multiple logins, you don't have to change your app. Existing Mobile SDK APIs work as before in the single-user scenario.

Mobile SDK assumes that each user account is unrelated to any other authenticated user account. Accordingly, Mobile SDK isolates data associated with each account from that of all others, thus preventing the mixing of data between accounts. Data isolation protects `SharedPreferences` files, SmartStore databases, `AccountManager` data, and any other flat files associated with an account.

Example: For native Android, the `RestExplorer` sample app demonstrates multi-user switching:

For native iOS, the `RestAPIExplorer` sample app demonstrates multi-user switching:

The following hybrid sample apps demonstrate multi-user switching:

- **Without SmartStore:** `ContactExplorer`
- **With SmartStore:** `AccountEditor`

Implementing Multi-User Support

Mobile SDK provides APIs for enabling multi-user support in native Android, native iOS, and hybrid apps.

Although Mobile SDK implements the underlying functionality, multi-user switching isn't initialized at runtime unless and until your app calls one of the following APIs:

Android native (`UserAccountManager` class methods)

```
public void switchToUser(UserAccount user)
```

```
public void switchToNewUser()
```

iOS native (`SFUserAccountManager` class methods)

```
- (void)switchToUser:(SFUserAccount *)newCurrentUser
```

```
- (void)switchToNewUser
```

Hybrid (JavaScript method)

```
switchToUser
```

To let the user switch to a different account, launch a selection screen from a button, menu, or some other control in your user interface. Mobile SDK provides a standard multi-user switching screen that displays all currently authenticated accounts in a radio button list. You can choose whether to customize this screen or just show the default version. When the user makes a selection, call the Mobile SDK method that launches the multi-user flow.

Before you begin to use the APIs, it's important that you understand the division of labor between Mobile SDK and your app. The following lists show tasks that Mobile SDK performs versus tasks that your app is required to perform in multi-user contexts. In particular, consider how to manage:

- Push Notifications (if your app supports them)
- SmartStore Soups (if your app uses SmartStore)
- Account Management

Push Notifications Tasks

Mobile SDK (for all accounts):

- Registers push notifications at login
- Unregisters push notifications at logout
- Delivers push notifications

Your app:

- Differentiates notifications according to the target user account
- Launches the correct user context to display each notification

SmartStore Tasks

Mobile SDK (for all accounts):

- Creates a separate SmartStore database for each authenticated user account
- Switches to the correct backing database each time a user switch occurs

Your app:

- Refreshes its cached credentials, such as instances of SmartStore held in memory, after every user switch or logout

Account Management Tasks

Mobile SDK (for all accounts):

- Loads the correct account credentials every time a user switch occurs

Your app:

- Refreshes its cached credentials, such as authenticated REST clients held in memory, after every user switch or logout

Android Native APIs

Native classes in Mobile SDK for Android do most of the work for multi-user support. Your app makes a few simple calls and handles any data cached in memory. You also have the option of customizing the user switching activity.

To support user switching, Mobile SDK for Android defines native classes in the `com.salesforce.androidsdk.accounts`, `com.salesforce.androidsdk.ui`, and `com.salesforce.androidsdk.util` packages. Classes in the `com.salesforce.androidsdk.accounts` package include:

- `UserAccount`
- `UserAccountManager`

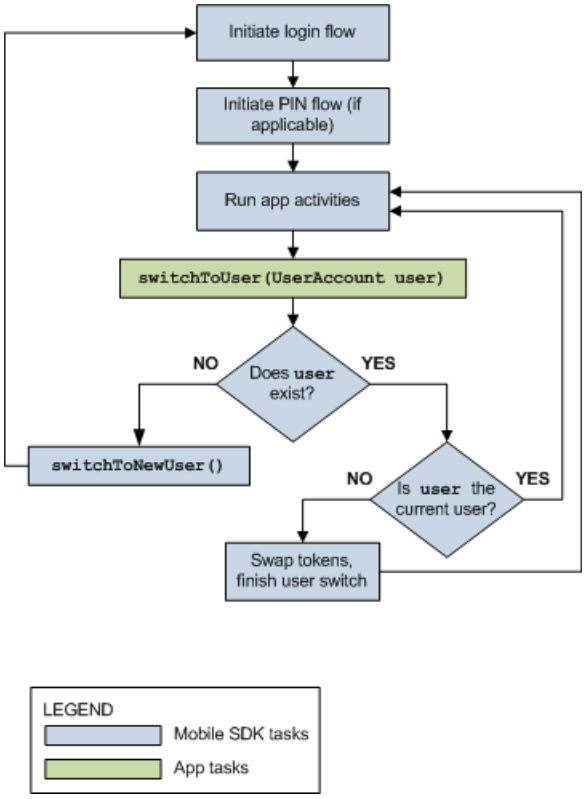
The `com.salesforce.androidsdk.ui` package contains the `AccountSwitcherActivity` class. You can extend this class to add advanced customizations to the account switcher activity.

The `com.salesforce.androidsdk.util` package contains the `UserSwitchReceiver` abstract class. You must implement this class if your app caches data other than tokens.

The following sections briefly describe these classes. For full API reference documentation, see https://forcedotcom.github.io/SalesforceMobileSDK-Android/index.html.

Multi-User Flow

For native Android apps, the `UserAccountManager.switchToUser()` Mobile SDK method launches the multi-user flow. Once your app calls this method, the Mobile SDK core handles the execution flow through all possible paths. The following diagram illustrates this flow.



IN THIS SECTION:

UserAccount Class

The `UserAccount` class represents a single user account that is currently authenticated. It encapsulates data that can be used to uniquely identify a user account.

UserAccountManager Class

The `UserAccountManager` class provides methods to access authenticated accounts, add new accounts, log out existing accounts, and switch between existing accounts.

AccountSwitcherActivity Class

Use or extend the `AccountSwitcherActivity` class to display the user switching interface.

UserSwitchReceiver Class

If your native Android app caches data other than tokens, implement the `UserSwitchReceiver` abstract class to receive notifications of user switching events.

UserAccount Class

The `UserAccount` class represents a single user account that is currently authenticated. It encapsulates data that can be used to uniquely identify a user account.

Constructors

You can create `UserAccount` objects directly, from a JSON object, or from a bundle.

Constructor	Description
`public UserAccount( String authToken, String refreshToken, String loginServer, String idUrl, String instanceServer, String orgId, String userId, String username, String accountName, String clientId, String communityId, String communityUrl )`	Creates a `UserAccount` object using values you specify.
`public UserAccount(JSONObject object)`	Creates a `UserAccount` object from a JSON string.
`public UserAccount(Bundle bundle)`	Creates a `UserAccount` object from an Android application bundle.

Methods

Method	Description
`public String getOrgLevelStoragePath()`	Returns the organization level storage path for this user account, relative to the higher level directory of app data. The higher level directory could be `files`. The output is in the format `/{orgID}/`. This storage path is meant for data that can be shared across multiple users of the same organization.
`public String getUserLevelStoragePath()`	Returns the user level storage path for this user account, relative to the higher level directory of app data. The higher level directory could be `files`. The output is in the format `/{orgID}/{userID}/`. This storage path is meant for data that is unique to a particular user in an organization, but common across all the communities that the user is a member of within that organization.
`public String getCommunityLevelStoragePath(String communityId)`	Returns the community level storage path for this user account, relative to the higher level directory of app data. The higher level directory could be `files`. The output is in the format `/{orgID}/{userID}/{communityID}/`. If `communityID` is null and then the output would be `/{orgID}/{userID}/internal/`. This storage path is

Method	Description
	meant for data that is unique to a particular user in a specific community.
`public String getOrgLevelFilenameSuffix()`	Returns a unique suffix for this user account, that can be appended to a file to uniquely identify this account, at an organization level. The output is in the format `_{orgID}`. This suffix is meant for data that can be shared across multiple users of the same organization.
`public String getUserLevelFilenameSuffix()`	Returns a unique suffix for this user account, that can be appended to a file to uniquely identify this account, at a user level. The output is in the format `_{orgID}_{userID}`. This suffix is meant for data that is unique to a particular user in an organization, but common across all the communities that the user is a member of within that organization.
`public String getCommunityLevelFilenameSuffix(String communityId)`	Returns a unique suffix for this user account, that can be appended to a file to uniquely identify this account, at a community level. The output is in the format `_{orgID}_{userID}_{communityID}`. If `communityID` is null and then the output would be `_{orgID}_{userID}_internal`. This suffix is meant for data that is unique to a particular user in a specific community.

UserAccountManager Class

The `UserAccountManager` class provides methods to access authenticated accounts, add new accounts, log out existing accounts, and switch between existing accounts.

You don't directly create instances of `UserAccountManager`. Instead, obtain an instance using the following call:

```
SalesforceSDKManager.getInstance().getUserAccountManager();
```

Methods

Method	Description
`public UserAccount getCurrentUser()`	Returns the currently active user account.
`public List<UserAccount> getAuthenticatedUsers()`	Returns the list of authenticated user accounts.
`public boolean doesUserAccountExist(UserAccount account)`	Checks whether the specified user account is already authenticated.
`public void switchToUser(UserAccount user)`	Switches the application context to the specified user account. If the specified user account is invalid or null, this method launches the login flow.
`public void switchToNewUser()`	Launches the login flow for a new user to log in.

Method	Description
`public void signoutUser(UserAccount userAccount, Activity frontActivity)`	Logs the specified user out of the application and wipes the specified user's credentials.

AccountSwitcherActivity Class

Use or extend the `AccountSwitcherActivity` class to display the user switching interface.

The `AccountSwitcherActivity` class provides the screen that handles multi-user logins. It displays a list of existing user accounts and lets the user switch between existing accounts or sign into a new account. To enable multi-user logins, launch the activity from somewhere in your app using the following code:

```
final Intent i = new Intent(this, SalesforceSDKManager.getInstance().
    getAccountSwitcherActivityClass());
i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
this.startActivity(i);
```

For instance, you might launch this activity from a "Switch User" button in your user interface. See `SampleApps/RestExplorer` for an example.

If you like, you can customize and stylize `AccountSwitcherActivity` through XML.

For more control, you can extend `AccountSwitcherActivity` and replace it with your own custom sub-class. To replace the default class, call `SalesforceSDKManager.setAccountSwitcherActivityClass()`. Pass in a reference to the class file of your replacement activity class, such as `AccountSwitcherActivity.class`.

UserSwitchReceiver Class

If your native Android app caches data other than tokens, implement the `UserSwitchReceiver` abstract class to receive notifications of user switching events.

Every time a user switch occurs, Mobile SDK broadcasts an intent. The intent action is declared in the `UserAccountManager` class as:

```
public static final String USER_SWITCH_INTENT_ACTION =
    "com.salesforce.USERSWITCHED";
```

This broadcast event gives applications a chance to properly refresh their cached resources to accommodate user switching. To help apps listen for this event, Mobile SDK provides the `UserSwitchReceiver` abstract class. This class is implemented in the following Salesforce activity classes:

- `SalesforceActivity`
- `SalesforceListActivity`
- `SalesforceExpandableListActivity`

If your main activity extends one of the Salesforce activity classes, you don't need to implement `UserSwitchReceiver`.

If you've cached only tokens in memory, you don't need to do anything—Mobile SDK automatically refreshes tokens.

If you've cached user data other than tokens, override your activity's `refreshIfUserSwitched()` method with your custom refresh actions.

If your main activity does not extend one of the Salesforce activity classes, implement `UserSwitchReceiver` to handle cached data during user switching.

To set up the broadcast receiver:

1. Implement a subclass of `UserSwitchReceiver`.
2. Register your subclass as a receiver in your activity's `onCreate()` method.
3. Unregister your receiver in your activity's `onDestroy()` method.

For an example, see the `ExplorerActivity` class in the `RestExplorer` sample application.

If your application is a hybrid application, no action is required.

The `SalesforceDroidGapActivity` class refreshes the cache as needed when a user switch occurs.

Methods

A single method requires implementation.

Method Name	Description
`protected abstract void onUserSwitch();`	Implement this method to handle cached user data (other than tokens) when user switching occurs.

iOS Native APIs

Native classes in Mobile SDK for iOS do most of the work for multi-user support. Your app makes a few simple calls and handles any data cached in memory. You also have the option of customizing the user switching activity.

To support user switching, Mobile SDK for iOS defines native classes in the `Security` folder of the `SalesforceSDKCore` library. Classes include:

- `SFUserAccount`
- `SFUserAccountManager`

The following sections briefly describe these classes. For full API reference documentation, see SalesforceSDKCore Reference.

IN THIS SECTION:

SFUserAccount Class

The `SFUserAccount` class represents a single user account that's currently authenticated. It encapsulates data that can be used to uniquely identify a user account.

SFUserAccountManager Class

The `SFUserAccountManager` class provides methods to access authenticated accounts, add new accounts, log out accounts, and switch between accounts.

SFUserAccount Class

The `SFUserAccount` class represents a single user account that's currently authenticated. It encapsulates data that can be used to uniquely identify a user account.

Properties

You can create `SFUserAccount` objects directly, from a JSON object, or from a bundle.

Property	Description
`@property (nonatomic, copy) NSSet *accessScopes`	The access scopes for this user.
`@property (nonatomic, strong) SFOAuthCredentials *credentials;`	The credentials that are associated with this user.
`@property (nonatomic, strong) SFIdentityData *idData;`	The identity data that's associated with this user.
`@property (nonatomic, copy, readonly) NSURL *apiUrl;`	The URL that can be used to invoke any API on the server side. This URL takes into account the current community if available.
`@property (nonatomic, copy) NSString *email;`	The user's email address.
`@property (nonatomic, copy) NSString *organizationName;`	The name of the user's organization.
`@property (nonatomic, copy) NSString *fullName;`	The user's first and last names.
`@property (nonatomic, copy) NSString *userName;`	The user's username.
`@property (nonatomic, strong) UIImage *photo;`	The user's photo, typically a thumbnail of the user. The consumer of this class must set this property at least once in order to use the photo. This class doesn't fetch the photo from the server; it stores and retrieves the photo locally.
`@property (nonatomic) SFUserAccountAccessRestriction accessRestrictions;`	The access restrictions that are associated with this user.
`@property (nonatomic, copy) NSString *communityId;`	The current community ID, if the user is logged into a community. Otherwise, this property is nil.
`@property (nonatomic, readonly, getter = isSessionValid) BOOL sessionValid;`	Returns YES if the user has an access token and, presumably, a valid session.

Property	Description
`@property (nonatomic, copy) NSDictionary *customData;`	The custom data for the user. Because this data can be serialized, the objects that are contained in `customData` must follow the `NSCoding` protocol.

Global Function

Function Name	Description
`NSString *SFKeyForUserAndScope (SFUserAccount *user, SFUserAccountScope scope);`	Returns a key that uniquely identifies this user account for the given scope. If you set `scope` to `SFUserAccountScopeGlobal`, the same key will be returned regardless of the user account.

SFUserAccountManager Class

The `SFUserAccountManager` class provides methods to access authenticated accounts, add new accounts, log out accounts, and switch between accounts.

To access the singleton `SFUserAccountManager` instance, send the following message:

```
[SFUserAccountManager sharedInstance]
```

Properties

Property	Description
`@property (nonatomic, strong) SFUserAccount *currentUser`	The current user account. If the user has never logged in, this property may be nil.
`@property (nonatomic, readonly) NSString *currentUserId`	A convenience property to retrieve the current user's ID. This property is an alias for `currentUser.credentials.userId`.
`@property (nonatomic, readonly) NSString *currentCommunityId`	A convenience property to retrieve the current user's community ID. This property is an alias for `currentUser.communityId`.
`@property (nonatomic, readonly) NSArray *allUserAccounts`	An `NSArray` of all the `SFUserAccount` instances for the app.
`@property (nonatomic, readonly) NSArray *allUserIds`	Returns an array that contains all user IDs.
`@property (nonatomic, copy) NSString *activeUserId`	The most recently active user ID. If the user that's specified by `activeUserId` is removed from the accounts list, this user may be temporarily different from the current user.
`@property (nonatomic, strong) NSString *loginHost`	The host to be used for login.

Property	Description
`@property (nonatomic, assign) BOOL retryLoginAfterFailure`	A flag that controls whether the login process restarts after it fails. The default value is YES.
`@property (nonatomic, copy) NSString *oauthCompletionUrl`	The OAuth callback URL to use for the OAuth login process. Apps can customize this property. By default, the property's value is copied from the `SFDCOAuthRedirectUri` property in the main bundle. The default value is `@"testsfdc:///mobilesdk/detect/oauth/done"`.
`@property (nonatomic, copy) NSSet *scopes`	The OAuth scopes that are associated with the app.

Methods

Method	Description
`- (NSString*) userAccountPlistFileForUser:(SFUserAccount*)user`	Returns the path of the `.plist` file for the specified user account.
`- (void) addDelegate:(id<SFUserAccountManagerDelegate>)delegate`	Adds a delegate to this user account manager.
`- (void) removeDelegate:(id<SFUserAccountManagerDelegate>)delegate`	Removes a delegate from this user account manager.
`- (SFLoginHostUpdateResult*)updateLoginHost`	Sets the app-level login host to the value in app settings.
`- (BOOL)loadAccounts:(NSError**)error`	Loads all accounts.
`- (SFUserAccount*)createUserAccount`	Can be used to create an empty user account if you want to configure all of the account information yourself. Otherwise, use `[SFAuthenticationManager loginWithCompletion:failure:]` to automatically create an account when necessary.
`- (SFUserAccount*) userAccountForUserId:(NSString*)userId`	Returns the user account that's associated with a given user ID.
`- (NSArray*) accountsForOrgId:(NSString*)orgId`	Returns all accounts that have access to a particular organization.

Method	Description
`- (NSArray *) accountsForInstanceURL:(NSString *)instanceURL`	Returns all accounts that match a particular instance URL.
`- (void)addAccount:(SFUserAccount *)acct`	Adds a user account.
`- (BOOL) deleteAccountForUserId:(NSString*)userId error:(NSError **)error`	Removes the user account that's associated with the given user ID.
`- (void)clearAllAccountState`	Clears the account's state in memory (but doesn't change anything on the disk).
`- (void) applyCredentials: (SFOAuthCredentials*)credentials`	Applies the specified credentials to the current user. If no user exists, a user is created.
`- (void)applyCustomDataToCurrentUser: (NSDictionary*)customData`	Applies custom data to the `SFUserAccount` that can be accessed outside that user's sandbox. This data persists between app launches. Because this data will be serialized, make sure that objects that are contained in `customData` follow the `NSCoding` protocol. **Important:** Use this method only for nonsensitive information.
`- (void)switchToNewUser`	Switches from the current user to a new user context.
`- (void)switchToUser:(SFUserAccount *)newCurrentUser`	Switches from the current user to the specified user account.
`- (void) userChanged:(SFUserAccountChange)change`	Informs the `SFUserAccountManager` object that something has changed for the current user.

Hybrid APIs

Hybrid apps can enable multi-user support through Mobile SDK JavaScript APIs. These APIs reside in the `SFAccountManagerPlugin` Cordova-based module.

SFAccountManagerPlugin Methods

Before you call any of these methods, you need to load the `sfaccountmanager` plug-in. For example:

```
cordova.require("com.salesforce.plugin.sfaccountmanager").logout();
```

Method Name	Description
`getUsers`	Returns the list of users already logged in.
`getCurrentUser`	Returns the current active user.
`logout`	Logs out the specified user if a user is passed in, or the current user if called with no arguments.
`switchToUser`	Switches the application context to the specified user, or launches the account switching screen if no user is specified.

Hybrid apps don't need to implement a receiver for the multi-user switching broadcast event. This handler is implemented by the `SalesforceDroidGapActivity` class.

CHAPTER 21 Mobile SDK Tools for Developers

In this chapter ...

Mobile SDK provides tools that help developers see what's happening in their apps at runtime.

For instance, you can inspect and query SmartStore soups, browse authenticated users, and view the user agent string. These tools are designed primarily for debugging purposes. In most cases, you disable this information in production builds.

In-App Developer Support

Mobile SDK 6.0 introduces several new screens in native apps for debugging during app development. These features usually do not require coding and are designed for debug builds.

The Dev Support dialog box is the launchpad for all available support screens. The dialog box presents only the options that are pertinent to the type of app you're running. During debugging, you can access the Dev Support through a keyboard shortcut or gesture.

By default, these tools are enabled in debug builds only. However, if necessary, you can use an SDK call to enable or disable the tools in production builds.

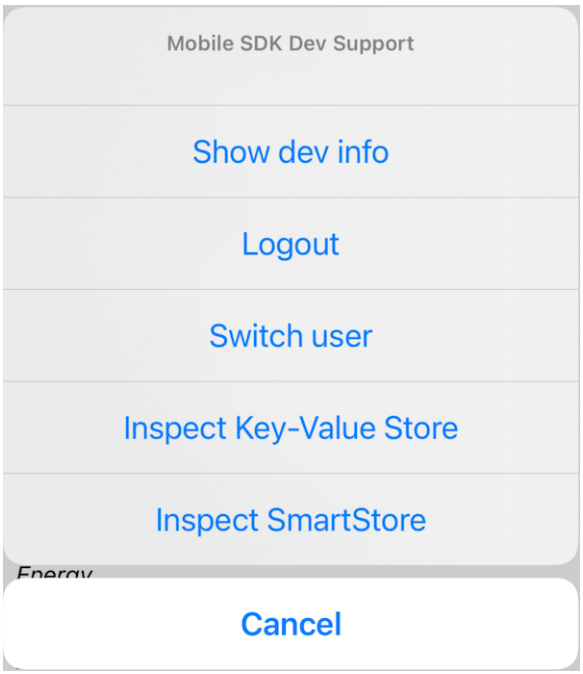
Launch the Dev Support Dialog Box

iOS

To launch the Dev Support dialog box, use one of the following options.

- Shake your physical device.
- From the iOS Simulator menu, select **Hardware** > **Shake Gesture**.
- When your app is running in the iOS Simulator, use the `^+Command+z` keyboard shortcut.

Here's the iOS screen.

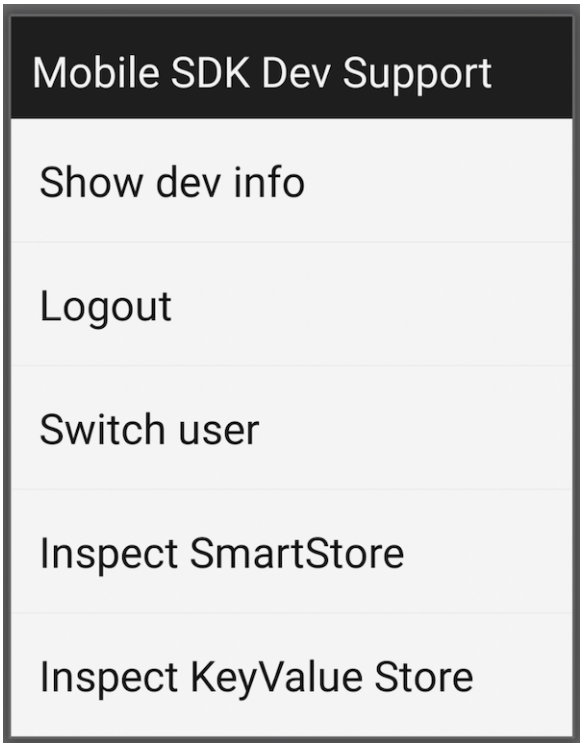


Android

To launch the Dev Support dialog box, use one of the following options.

- When your app is running in the Android emulator, use the `Command+m` (Mac) or `Ctrl+m` (Windows) keyboard shortcut.
- In a system command shell, run: `adb shell input keyevent 82`

Here's the Android screen.



Enable Dev Support Tools in Production Builds

If you require the Developer Support tools in production builds, set the property that indicates Dev Support is enabled. This use case is unusual.

iOS

```
// To enable
[SalesforceSDKManager sharedManager].isDevSupportEnabled = YES;
// To disable
[SalesforceSDKManager sharedManager].isDevSupportEnabled = NO;
```

Android

```
// To enable
SalesforceSDKManager.getInstance().setDevSupportEnabled(true);
// To disable
SalesforceSDKManager.getInstance().setDevSupportEnabled(false);
```

Important: Don't forget to disable the feature before distributing your app to the public.

Support Options

The Dev Support dialog box presents options based on the app type. In every case you get:

- **Show dev info**—brings up the dev info screen.
- **Logout**—Logs out the current user. Useful when you're getting started and have not yet had time to add a logout button or action into your UI.

- **Switch user**—Switch to a different user. Useful when you are getting started and have not yet had time to add a switch user button or action into your UI.
- **Inspect Key-Value Store**—Lets you inspect values stored in the encrypted key-value store.

If your application uses SmartStore, you also see:

Inspect SmartStore—Displays the SmartStore inspector screen.

SmartStore inspector is useful during development because it decrypts the data for display purposes. This decryption applies only to the inspector display—data remains encrypted on disk.

If your application is a React Native application, you also see:

React native dev support—The React Native dev menu.

Dev Info Screen

The Dev Info screen shows a collection of information about the app, its configuration, and so on, based on the app type. In every case you get:

- SDK version
- App type
- User agent
- Native browser for login enabled (indicates whether advanced authentication is configured)
- Identity Provider login enabled (indicates whether the app can use another app to provide login)
- Current user
- Authenticated users
- Boot config settings
- Managed (indicates whether the app is managed)
- Managed preferences (app settings pushed to the app by the MDM provider, if applicable)

For SmartStore apps, you get the following additional information:

- SQLCipher version
- SQLCipher compile options
- Names of user stores
- Names of global stores

SmartStore Inspector

The SmartStore Inspector screen is a legacy feature that lets user see a list of SmartStore soups and their indices, and run custom queries. In Mobile SDK 6.0, this screen adds a store picker that lets you choose which global or user store to inspect.

Extending the Dev Info Screen

The Dev Info screen gets the information shown in its screen from a method named `getDevSupportInfos` in `SalesforceSDKManager`. Some subclasses of `SalesforceSDKManager` override this method. To show your own custom information, you can also override it. To do so, implement a subclass of `SalesforceSDKManager`, override the method, and use it in your application.

iOS

```
/**
  * @return Array of name1, value1, name2, value2, etc.) to show in
SFSDKDevInfoViewController
  */
- (NSArray*) getDevSupportInfos;
```

Android

```
/**
  * @return Dev info (list of name1, value1, name2, value2, etc.) to show in DevInfoActivity

  */
public List<String> getDevSupportInfos();
```

CHAPTER 22 Logging and Analytics

In this chapter ...

Mobile SDK uses analytics and logging for its own purposes and also provides a logging framework for apps. You can disable the analytics feature in your app, and you can use the logging framework to output messages for your custom components.

iOS Compiler-Level Logging

Mobile SDK 7.0 simplifies iOS logging and refers it to the underlying operating system framework.

To access the logging system, call the `os_log()` function. This function gives you access to the Apple unified logging system. If you like, you can also pass a custom component log object and set a log level. See https://developer.apple.com/documentation/os/logging for details.

Note: The Salesforce Logging Framework on iOS is not currently recommended for external use. If you have legacy code that uses `SFSDKLogger`, you can continue using it as follows:

1. In each source file that uses `SFSDKLogger`, replace

   ```
   #import <SalesforceAnalytics/SFSDKLogger.h>
   ```

   with

   ```
   #import <SalesforceSDKCommon/SFLogger.h>
   ```

2. Using Xcode Refactor, replace all instances of `SFSDKLogger` with `SFLogger`.

Example: You can replace `SalesforceLogger` calls in the Swift forceios template as follows. These simplistic examples use the default component logger to log debug messages in the Xcode console.

```
RestClient.shared.send(request: request)
{ [weak self] (result) in
    switch result {
        case .success(let response):
            self?.handleSuccess(response: response, request: request)
        case .failure(let error):
            // SalesforceLogger.d(RootViewController.self,
            //    message:"Error invoking: \(request) , \(error)")
            os_log("\nError invoking: %@", log: .default, type: .debug, request)
    }
}
```

Android Logging Framework

Mobile SDK provides a logging framework that allows developers to easily create logs for app components.

With this framework, you can

- Output logs to both console and file.
- Output logs for your custom components and Mobile SDK components.
- Log at a component level. For example, Mobile Sync and SmartStore use separate loggers. You can also create custom loggers for your own components.
- Set logging levels, such as ERROR, DEBUG, and VERBOSE.
- Configure logging levels per component. For example, you can set Mobile Sync to the ERROR level and SmartStore to VERBOSE.

Using the logging framework is like playing tunes on a juke box. To obtain and use a logger instance:

1. **Browse the tunes**—Obtain a list of your app's available components. This runtime-only step isn't necessary if you already know the name of the component.

2. **Make your selection**—Pass the name of a component to the framework. To "play" your selection, Mobile SDK returns a logger instance for the chosen component. It doesn't matter whether the logger exists—Mobile SDK creates one if necessary.
3. **Bust a move**—Set the logging level, write a line to the log, limit the number of log lines, turn logging off or on.

Not a perfect analogy, but hopefully you get the point. To get the details, continue reading.

Create a Logger for Your Component

A component represents a virtual domain in your app's functionality. The component can represent your entire app or just a portion of its feature set. You devise a name for the component when you create the logger. You can then use the logger to document any notable conditions arising in the component's domain.

To create a logger for your own component, you don't have to override, extend, or implement anything. Instead, you request a logger from the logging framework using the name of your component. If the logger isn't already in the components list, the logging framework creates a singleton logger and adds the component to the list. Otherwise, it returns the existing logger. All threads share this logger.

You can create a single logger or multiple loggers. For example, to log all messages from your app at the same level, create a single logger that uses your app name. One logger is often sufficient for an app. However, if you want to differentiate logging levels between various parts of your app, you can create more loggers with component-based names. Be careful not to go overboard—you don't want excessive logging activity to degrade your app's performance.

You request loggers through the following objects.

- **Android:** `com.salesforce.androidsdk.analytics.logger.SalesforceLogger`

Use the following APIs to get loggers for custom components or standard Mobile SDK components.

Android

```
public synchronized static SalesforceLogger getLogger(String componentName, Context context);
```

For example:

```
SalesforceLogger.getLogger(”MyComponent”, context);
```

Get All Components

The following `SalesforceLogger` method returns a list of the names of all components that are currently associated with a logger. If you don't know the names of Mobile SDK internal components, your code can discover them at runtime in this list. You can use this list, for instance, to turn off logging for all components. How your app uses this information is up to your business logic.

Android

```
public synchronized static Set<String> getComponents();
```

Customize Logger Output

Once you get a logger instance, you can control the type and quantity of information the logger outputs.

- **Set a Component's Log Level**

  Each app component can have its own logger instance and its own log level. The default log level is DEBUG for a debug build and ERROR for a release build. Use the following APIs to set a component's log levels.

  Android

  ```
  public void setLogLevel(Level level);
  ```

Android loggers use an internal public enum that mirrors the Android default log levels. For example:

```
logger.setLogLevel(Level.INFO);
```

- **Write a Log Line**

  The following APIs can be used to write a log line using the new framework. The log line is automatically written to both console and file, unless file logging is disabled for that component.

  Android

  ```
  public void log(Level level, String tag, String message);
  ```

  ```
  public void log(Level level, String tag, String message, Throwable e);
  ```

- **Enable or Disable File Logging**

  The logging framework logs messages to both console and file by default. File logging can be disabled if necessary, using the following APIs.

  Android

  ```
  public synchronized void disableFileLogging();
  ```

  Enable file logging:

  ```
  public synchronized void enableFileLogging(int maxSize);
  ```

  In this API, `maxSize` represents the maximum number of log lines that the file can hold before the log lines are rolled.

Mobile SDK Logging Components

Mobile SDK provides a default logger for each of its standard components. This architecture allows you to control the log level of each component independently of other components. For example, you could set the log level of Mobile Sync to INFO while the log level of SmartStore is set to ERROR.

Here are lists of the standard component loggers in Mobile SDK.

Android

- `SalesforceAnalyticsLogger`
- `SalesforceSDKLogger`
- `SmartStoreLogger`
- `MobileSyncLogger`
- `SalesforceReactLogger`
- `SalesforceHybridLogger`

Each of these logging components has convenience methods for adjusting log levels and logging messages associated with those core components.

Instrumentation and Event Collection

Mobile SDK 5.0 introduces a new framework that adds analytical instrumentation to Mobile SDK apps. Through this instrumentation, apps collect event data that describe how consuming apps use Mobile SDK. Mobile SDK periodically uploads logs of these events to the Salesforce cloud. This information helps us focus on the features that matter most to your customers. We do not collect any data specific to users or their Salesforce organizations.

Mobile SDK app users and developers do not have access to the information Salesforce gathers. Salesforce collects it solely for its own use. The software that collects the data is maintained in Mobile SDK open source repos at github.com/forcedotcom.

Mobile SDK 5.0 and later enable instrumentation by default. Mobile SDK automatically publishes collected framework events to the Salesforce cloud on the following schedule:

- **iOS:** When the app goes to the background.
- **Android:** Every 8 hours.

Your app can toggle event logging on or off. On Android, your app can also change the collection upload frequency.

To manage the event logging service, use the following APIs. You call each API on an instance of an analytics manager object, which you initialize with your app's current user account.

Toggle Event Logging

Android

For Android, call the `enableLogging(boolean enabled)` method.

```
final UserAccount curAccount = UserAccountManager.getInstance().getCurrentUser();
final SalesforceAnalyticsManager sfAnalyticsManager =
    SalesforceAnalyticsManager.getInstance(curAccount);
sfAnalyticsManager.enableLogging(false);
```

iOS

For iOS, set the `BOOL loggingEnabled` property.

```
SFUserAccount *account = [SFUserAccountManager
sharedInstance].currentUser;
SFSDKSalesforceAnalyticsManager *sfAnalyticsManager =
    [SFSDKSalesforceAnalyticsManager
sharedInstanceWithUser:account];
sfAnalyticsManager.loggingEnabled = NO;
```

Check Event Logging Status

Android

For Android, call the `isLoggingEnabled(boolean enabled)` method.

```
final UserAccount curAccount = UserAccountManager.getInstance().getCurrentUser();
final SalesforceAnalyticsManager sfAnalyticsManager =
    SalesforceAnalyticsManager.getInstance(curAccount);
boolean enabled = sfAnalyticsManager.isLoggingEnabled();
```

iOS

For iOS, check the `BOOL isLoggingEnabled` property.

```
SFUserAccount *account = [SFUserAccountManager sharedInstance].currentUser;
SFSDKSalesforceAnalyticsManager *sfAnalyticsManager =
    [SFSDKSalesforceAnalyticsManager sharedInstanceWithUser:account];
BOOL enabled = sfAnalyticsManager.isLoggingEnabled;
```

Set Upload Frequency (Android Only)

On Android, you can set the frequency, in hours, of event log uploads. The default value is 8.

```
final UserAccount curAccount = UserAccountManager.getInstance().getCurrentUser();
final SalesforceAnalyticsManager sfAnalyticsManager =
    SalesforceAnalyticsManager.getInstance(curAccount);
sfAnalyticsManager.setPublishFrequencyInHours(numHours); // numHours is the desired upload
 interval in hours.
```

CHAPTER 23 Migrating from the Previous Release

In this chapter ...

If you're upgrading an app built with Salesforce Mobile SDK 11.0, follow these instructions to update your app to 11.1.

If you're upgrading an app that's built with a version earlier than Salesforce Mobile SDK 11.0, start upgrading with Migrating from Earlier Releases.

Migrate All Apps from 11.0 to 11.1

Mobile SDK 11.1 migration is easiest if you use the Salesforce CLI plugin or the Mobile SDK npm scripts.

Before you begin upgrading, read about new 11.1 features in What's New in Mobile SDK 11.1 on page 10. Mobile SDK 11.1 requires no code changes.

Native iOS (Swift, Objective-C)

- Make sure that you've installed the supported versions of iOS and Xcode. See iOS Basic Requirements, or use the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Choose one of the following options.
  - (Recommended) **Use the Salesforce CLI Mobile SDK plugin or the forceios npm script:** Recreate your app. For Saleforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:ios:create help
    ```

    For forceios, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
  - **Use CocoaPods:** If you created your app manually using CocoaPods, see Refreshing Mobile SDK Pods.
  - **Do it manually:** Manually create a new native template app in Swift. Follow the instructions in Creating an iOS Swift Project Manually.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs deprecated for future removal, and address any items that affect your codebase. See iOS Current Deprecations.
- After a successful build, check compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

Native Android (Java, Kotlin)

- Make sure that you've installed the supported versions of Android SDK and Android Studio. See Native Android Requirements, or use the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Choose one of the following options.
  - (Recommended) **Use the Salesforce CLI Mobile SDK plugin or forcedroid npm script:** Recreate your app. For Saleforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:android:create help
    ```

    For forcedroid, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
  - **Use Maven:** If you created your app manually using Maven, see Using Maven to Update Mobile SDK Libraries in Android Apps.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs deprecated for future removal, and address any items that affect your codebase until your build succeeds. See Android Current Deprecations.
- After a successful build, check the compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

React Native

- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcereact npm script. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:reactnative:create help
  ```

  For forcereact, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

Hybrid

- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcehybrid npm script. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:hybrid:create help
  ```

  For forcehybrid, follow the instructions in "Updating Hybrid Apps" at Updating Mobile SDK Apps (5.0 and Later).
- After you've recreated your app:
  - Migrate your app's artifacts into the new template.
  - Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
  - Consider adopting new features. See What's New in Mobile SDK 11.1.

See Also

- Set Up Salesforce DX

Migrating from Earlier Releases

To migrate from versions older than the previous release, perform the code upgrade steps for each intervening release, starting at your current version.

Migrate All Apps from 10.2 to 11.0

Mobile SDK 11.0 migration is easiest if you use the Salesforce CLI plugin or the Mobile SDK npm scripts.

Before you begin upgrading, read about new 11.0 features in What's New in Mobile SDK 11.1 on page 10. Mobile SDK 11.0 requires no code changes.

Native iOS (Swift, Objective-C)

- Make sure that you've installed the supported versions of iOS and Xcode. See iOS Basic Requirements, or use the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Choose one of the following options.

- (Recommended) **Use the Salesforce CLI Mobile SDK plugin or the forceios npm script:** Recreate your app. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:ios:create help
  ```

  For forceios, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
- **Use CocoaPods:** If you created your app manually using CocoaPods, see Refreshing Mobile SDK Pods.
- **Do it manually:** Manually create a new native template app in Swift. Follow the instructions in Creating an iOS Swift Project Manually.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs deprecated for future removal, and address any items that affect your codebase. See iOS Current Deprecations.
- After a successful build, check compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

Native Android (Java, Kotlin)

- Make sure that you've installed the supported versions of Android SDK and Android Studio. See Native Android Requirements, or use the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Choose one of the following options.
  - (Recommended) **Use the Salesforce CLI Mobile SDK plugin or forcedroid npm script:** Recreate your app. For Saleforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:android:create help
    ```

    For forcedroid, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
  - **Use Maven:** If you created your app manually using Maven, see Using Maven to Update Mobile SDK Libraries in Android Apps.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs deprecated for future removal, and address any items that affect your codebase until your build succeeds. See Android Current Deprecations.
- After a successful build, check the compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

React Native

- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcereact npm script. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:reactnative:create help
  ```

  For forcereact, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

Hybrid

- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcehybrid npm script. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:hybrid:create help
  ```

  For forcehybrid, follow the instructions in "Updating Hybrid Apps" at Updating Mobile SDK Apps (5.0 and Later).
- After you've recreated your app:
  - Migrate your app's artifacts into the new template.
  - Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
  - Consider adopting new features. See What's New in Mobile SDK 11.1.

See Also

- Set Up Salesforce DX

Migrate All Apps from 10.1 to 10.2

Mobile SDK 10.2 migration is easiest if you use the Salesforce CLI plugin or the Mobile SDK npm scripts.

Before you begin upgrading, read about new 10.2 features in What's New in Mobile SDK 11.1 on page 10. Mobile SDK 10.2 requires no code changes.

Native iOS (Swift, Objective-C)

- Make sure that you've installed the supported versions of iOS and Xcode. See iOS Basic Requirements, or use the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Choose one of the following options.
  - (Recommended) **Use the Salesforce CLI Mobile SDK plugin or the forceios npm script:** Recreate your app. For Saleforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:ios:create help
    ```

    For forceios, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
  - **Use CocoaPods:** If you created your app manually using CocoaPods, see Refreshing Mobile SDK Pods.
  - **Do it manually:** Manually create a new native template app in Swift. Follow the instructions in Creating an iOS Swift Project Manually.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs deprecated for future removal, and address any items that affect your codebase. See iOS Current Deprecations.
- After a successful build, check compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

Native Android (Java, Kotlin)

- Make sure that you've installed the supported versions of Android SDK and Android Studio. See Native Android Requirements, or use the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Choose one of the following options.
  - (Recommended) **Use the Salesforce CLI Mobile SDK plugin or forcedroid npm script:** Recreate your app. For Saleforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:android:create help
    ```

    For forcedroid, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
  - **Use Maven:** If you created your app manually using Maven, see Using Maven to Update Mobile SDK Libraries in Android Apps.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs deprecated for future removal, and address any items that affect your codebase until your build succeeds. See Android Current Deprecations.
- After a successful build, check the compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

React Native

- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcereact npm script. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:reactnative:create help
  ```

  For forcereact, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

Hybrid

- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcehybrid npm script. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:hybrid:create help
  ```

  For forcehybrid, follow the instructions in "Updating Hybrid Apps" at Updating Mobile SDK Apps (5.0 and Later).
- After you've recreated your app:
  - Migrate your app's artifacts into the new template.
  - Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
  - Consider adopting new features. See What's New in Mobile SDK 11.1.

See Also

- Set Up Salesforce DX

Migrate All Apps from 10.0 to 10.1

Mobile SDK 10.1 migration is easiest if you use the Salesforce CLI plugin or the Mobile SDK npm scripts.

Before you begin upgrading, read about new 10.1 features in What's New in Mobile SDK 11.1 on page 10. Mobile SDK 10.1 requires no code changes.

Native iOS (Swift, Objective-C)

- Make sure that you've installed the supported versions of iOS and Xcode. See iOS Basic Requirements, or use the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Choose one of the following options.
  - (Recommended) **Use the Salesforce CLI Mobile SDK plugin or the forceios npm script:** Recreate your app. For Saleforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:ios:create help
    ```

    For forceios, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
  - **Use CocoaPods:** If you created your app manually using CocoaPods, see Refreshing Mobile SDK Pods.
  - **Do it manually:** Manually create a new native template app in Swift. Follow the instructions in Creating an iOS Swift Project Manually.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs deprecated for future removal, and address any items that affect your codebase. See iOS Current Deprecations.
- After a successful build, check compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

Native Android (Java, Kotlin)

- Make sure that you've installed the supported versions of Android SDK and Android Studio. See Native Android Requirements, or use the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Choose one of the following options.
  - (Recommended) **Use the Salesforce CLI Mobile SDK plugin or forcedroid npm script:** Recreate your app. For Saleforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:android:create help
    ```

    For forcedroid, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
  - **Use Maven:** If you created your app manually using Maven, see Using Maven to Update Mobile SDK Libraries in Android Apps.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs deprecated for future removal, and address any items that affect your codebase until your build succeeds. See Android Current Deprecations.

- After a successful build, check the compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

React Native

- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcereact npm script. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:reactnative:create help
  ```

  For forcereact, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

Hybrid

- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcehybrid npm script. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:hybrid:create help
  ```

  For forcehybrid, follow the instructions in "Updating Hybrid Apps" at Updating Mobile SDK Apps (5.0 and Later).
- After you've recreated your app:
  - Migrate your app's artifacts into the new template.
  - Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
  - Consider adopting new features. See What's New in Mobile SDK 11.1.

See Also

- Set Up Salesforce DX

Migrate All Apps from 9.2 to 10.0

Mobile SDK 10.0 migration is easiest if you use the Salesforce CLI plugin or the Mobile SDK npm scripts.

Before you begin upgrading, read about new 10.0 features in What's New in Mobile SDK 11.1 on page 10.

Native iOS (Swift, Objective-C)

- Make sure that you've installed the supported versions of iOS and Xcode. See iOS Basic Requirements, or Supported Versions of Tools and Components for Mobile SDK 11.1.
- Choose one of the following options.
  - **Use the Salesforce CLI Mobile SDK plugin or the forceios npm script (recommended):** Recreate your app, and then migrate your app's artifacts into the new template. For Salesforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:ios:create help
    ```

    For forceios, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).

  - **Use CocoaPods:** If you created your app manually using CocoaPods, see Refreshing Mobile SDK Pods.
  - **Do it manually:** Manually create a new native template app in Swift, and then migrate your app's artifacts into the new template. Follow the instructions in Creating an iOS Swift Project Manually.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs removed in Mobile SDK 10.0, and address any items that affect your code base. See iOS APIs Removed in Mobile SDK 11.0.
- Review the list of APIs deprecated for future removal, and address any items that affect your code base. See iOS Current Deprecations.
- After a successful build, check compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

Native Android (Java, Kotlin)

- Make sure that you've installed the supported versions of Android SDK and Android Studio. See Native Android Requirements, or Supported Versions of Tools and Components for Mobile SDK 11.1.
- Choose one of the following options.
  - **Use the Salesforce CLI Mobile SDK plugin or forcedroid npm script (recommended):** Recreate your app with the plugin or script, and then migrate your app's artifacts into the new template. For Salesforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:android:create help
    ```

    For forcedroid, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
  - **Use Maven:** If you created your app manually using Maven, see Using Maven to Update Mobile SDK Libraries in Android Apps.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs removed in Mobile SDK 10.0, and address any items that affect your code base. See Android APIs Removed in Mobile SDK 11.0.
- Review the list of APIs deprecated for future removal, and address any items that affect your codebase until your build succeeds. See Android Current Deprecations.
- After a successful build, check the compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

React Native

Mobile SDK 10.0 requires no code changes.

- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See Supported Versions of Tools and Components for Mobile SDK 11.1.
- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcereact npm script. For Salesforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:reactnative:create help
  ```

  For forcereact help, type

  ```
  forcereact
  ```

- After you've recreated your app, migrate your app's artifacts into the new template.

Hybrid

Mobile SDK 10.0 requires no code changes.

- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See Supported Versions of Tools and Components for Mobile SDK 11.1.
- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcehybrid npm script. For Salesforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:hybrid:create help
  ```

  For forcehybrid, follow the instructions in "Updating Hybrid Apps" at Updating Mobile SDK Apps (5.0 and Later).

See Also

- Set Up Salesforce DX

Migrate All Apps from 9.1 to 9.2

Mobile SDK 9.2 migration is easiest if you use the Salesforce CLI plugin or the Mobile SDK npm scripts.

Before you begin upgrading, read about new 9.2 features in What's New in Mobile SDK 11.1 on page 10.

Native iOS (Swift, Objective-C)

- Make sure that you've installed the supported versions of iOS and Xcode. See iOS Basic Requirements, or use the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Choose one of the following options.
  - **Use the Salesforce CLI Mobile SDK plugin or the forceios npm script (recommended):** Recreate your app, and then migrate your app's artifacts into the new template. For Saleforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:ios:create help
    ```

    For forceios, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
  - **Use CocoaPods:** If you created your app manually using CocoaPods, see Refreshing Mobile SDK Pods.
  - **Do it manually:** Manually create a new native template app in Swift, and then migrate your app's artifacts into the new template. Follow the instructions in Creating an iOS Swift Project Manually.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs deprecated for future removal, and address any items that affect your code base. See iOS Current Deprecations.
- After a successful build, check compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

Key Store Encryption Upgrade

Mobile SDK 9.2 upgrades key store encryption. For most apps the upgrade happens automatically, but some apps require extra steps. See Upgrading Encryption in Apps on page 435.

Native Android (Java, Kotlin)

- Make sure that you've installed the supported versions of Android SDK and Android Studio. See Native Android Requirements, or use the Set Up Your Mobile SDK Developer Tools Trailhead project.

- Choose one of the following options.
  - **Use the Salesforce CLI Mobile SDK plugin or forcedroid npm script (recommended):** Recreate your app with the plugin or script, and then migrate your app's artifacts into the new template. For Saleforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:android:create help
    ```

    For forcedroid, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
  - **Use Maven:** If you created your app manually using Maven, see Using Maven to Update Mobile SDK Libraries in Android Apps.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs deprecated for future removal, and address any items that affect your codebase until your build succeeds. See Android Current Deprecations.
- After a successful build, check the compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

React Native

Mobile SDK 9.2 requires no code changes.

- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcereact npm script. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:reactnative:create help
  ```

  For forcereact, type

  ```
  forcereact create
  ```

  .
- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- After you've recreated your app, migrate your app's artifacts into the new template.

Hybrid

Mobile SDK 9.2 requires no code changes.

- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcehybrid npm script. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:hybrid:create help
  ```

  For forcehybrid, follow the instructions in "Updating Hybrid Apps" at Updating Mobile SDK Apps (5.0 and Later).

See Also

- Set Up Salesforce DX

Migrate All Apps from 9.0 to 9.1

Mobile SDK 9.1 migration is easiest if you use the Salesforce CLI plugin or the Mobile SDK npm scripts.

Before you begin upgrading, read about new 9.1 features in What's New in Mobile SDK 11.1 on page 10.

Native iOS (Swift, Objective-C)

- Make sure that you've installed the supported versions of iOS and Xcode. See iOS Basic Requirements, or use the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Choose one of the following options.
  - **Use the Salesforce CLI Mobile SDK plugin or the forceios npm script (recommended):** Recreate your app, and then migrate your app's artifacts into the new template. For Saleforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:ios:create help
    ```

    For forceios, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
  - **Use CocoaPods:** If you created your app manually using CocoaPods, see Refreshing Mobile SDK Pods.
  - **Do it manually:** Manually create a new native template app in Swift, and then migrate your app's artifacts into the new template. Follow the instructions in Creating an iOS Swift Project Manually.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs deprecated for future removal, and address any items that affect your codebase. See iOS Current Deprecations.
- After a successful build, check compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

Native Android (Java, Kotlin)

- Make sure that you've installed the supported versions of Android SDK and Android Studio. See Native Android Requirements, or use the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Choose one of the following options.
  - **Use the Salesforce CLI Mobile SDK plugin or forcedroid npm script (recommended):** Recreate your app with the plugin or script, and then migrate your app's artifacts into the new template. For Saleforce CLI, follow the instructions at the command line by typing

    ```
    sf mobilesdk:android:create help
    ```

    For forcedroid, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
  - **Use Maven:** If you created your app manually using Maven, see Using Maven to Update Mobile SDK Libraries in Android Apps.

After you've recreated your app:

- Migrate your app's artifacts into the new template.
- Review the list of APIs deprecated for future removal, and address any items that affect your codebase until your build succeeds. See Android Current Deprecations.
- After a successful build, check the compiler warnings for deprecations or other Mobile SDK issues you've missed.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

React Native

Mobile SDK 9.1 requires no code changes.

- Migrate your app's artifacts into the new template.
- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcereact npm script. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:reactnative:create help
  ```

  For forcereact, follow the instructions in "Updating Native and React Native Apps" at Updating Mobile SDK Apps (5.0 and Later).
- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Consider adopting new features. See What's New in Mobile SDK 11.1.

Hybrid

Mobile SDK 9.1 requires no code changes.

- Migrate your app's artifacts into the new template.
- Make sure that you've installed the supported versions of the mobile platforms you're targeting. See the Set Up Your Mobile SDK Developer Tools Trailhead project.
- Recreate your app with the Salesforce CLI Mobile SDK plug-in or the forcehybrid npm script. For Saleforce CLI, follow the instructions at the command line by typing

  ```
  sf mobilesdk:hybrid:create help
  ```

  For forcehybrid, follow the instructions in "Updating Hybrid Apps" at Updating Mobile SDK Apps (5.0 and Later).

See Also

- Set Up Salesforce DX

CHAPTER 24 Reference

In this chapter ...

Reference documentation is hosted on GitHub.

- For iOS:
  - *SalesforceAnalytics Library Reference* at https://forcedotcom.github.io/SalesforceMobileSDK-iOS/Documentation/SalesforceAnalytics/html/index.html
  - *SalesforceSDKCommon Library Reference* at https://forcedotcom.github.io/SalesforceMobileSDK-iOS/Documentation/SalesforceSDKCommon/html/index.html
  - *SalesforceSDKCore Library Reference* at https://forcedotcom.github.io/SalesforceMobileSDK-iOS/Documentation/SalesforceSDKCore/html/index.html
  - *SmartStore Library Reference* at https://forcedotcom.github.io/SalesforceMobileSDK-iOS/Documentation/SmartStore/html/index.html
  - *Mobile Sync Library Reference* at https://forcedotcom.github.io/SalesforceMobileSDK-iOS/Documentation/MobileSync/html/index.html
- For Android: https://forcedotcom.github.io/SalesforceMobileSDK-Android/index.html

Supported Salesforce APIs

The `RestRequest` class provides factory and extension methods that wrap Salesforce API calls. These methods use request parameters that you provide to construct the network call.

Swift, Java, Kotlin	Objective-C
`RestRequest`	`SFRestRequest`

`RestRequest` returns a specialized copy of itself that reflects your parameters. To send your request to Salesforce, you pass this customized object to the Mobile SDK REST client.

`RestRequest` supports the following Salesforce APIs.

IN THIS SECTION:

Describe Global

Returns a list of all available objects in your org and their metadata.

Metadata

Describes metadata provided by sObject basic information for the specified object .

Notification

Fetches a single notification by its notification ID.

Notification Update

Updates the “read” (if non-null) and “seen” (if non-null) statuses of the notification with the given ID.

Notifications

Gets the given number (maximum 20) of archived Notification Builder notifications based on the given “before” or “after” date.

Notifications Status

Get the status of the current user’s notifications, including unread and unseen count.

Notifications Update

Updates the “read” (if non-null) and “seen” (if non-null) statuses of notifications with the given IDs, or those sent before the given date.

Object Layout

Gets layout metadata for the specified object type and parameters.

Resources

Gets available resources for the specified API version, including resource name and URI.

Retrieve

Retrieves a single sObject record by object ID.

SOSL Search

Performs the given SOSL search.

Search Result Layout

Gets the search result layout for up to 100 objects with a single query.

Search Scope and Order

Gets an ordered list of objects in the current user’s default global search scope.

SObject Tree

Creates one or more sObject trees with root records of the specified object type.

SOQL Query

SOQL Query All

Executes the given SOQL string. The result includes all current and deleted objects that satisfy the query.

User Info

Returns information associated with the current user.

Update

Updates specified fields of the requested record with the given values. Can also prevent the update from occurring if the record has been modified since a given date.

Upsert

Updates or inserts an object from external data.

Versions

Gets summary information about each Salesforce API version currently available.

SEE ALSO:

Using Salesforce REST APIs with Mobile SDK

Using REST APIs

Batch Request

Executes a batch of subrequests.

Returns a `RestRequest` object containing a batch of up to 25 subrequests specified in a list of `RestRequest` objects. Each subrequest counts against rate limits.

Parameters

- `requests` (array/list)
- `haltOnError` (Boolean)
- `apiVersion` (string)

iOS

Swift

```
RestClient.shared.batchRequest(requests:haltOnError:apiVersion:)
```

Objective-C

```
- (SFRestRequest *) batchRequest:(NSArray<SFRestRequest *> *)requests
                     haltOnError:(BOOL)haltOnError
                      apiVersion:(nullable NSString *)apiVersion;
```

Android

Java

```
public static BatchRequest getBatchRequest(String apiVersion, boolean haltOnError,
List<RestRequest> requests) throws JSONException
```

See Also

- Batch and Composite Requests on page 90 (iOS)
- Batch and Composite Requests on page 141 (Android)
- "Composite Batch" in *REST API Developer Guide*

Briefcase Priming Records

Returns a request object that obtains record IDs from briefcases assigned to the connected app.

Mobile SDK provides a custom response object for parsing the results of Briefcase Priming Records requests. See Briefcase Priming Records Response.

Using Relay Tokens to Acquire Record IDs

Retrieve batches of record IDs in an iterative loop that's controlled by a relay token, as follows:

- In your first request, set `relayToken` to null. The response to this request will contain a new relay token.
- In each subsequent request, set `relayToken` to the relay token value of the previous response.

In your request, you can also set a timestamp to retrieve only records that changed after the given time.

Parameters

- API version (string)
- Relay token (string; can be null)
- "Changed after" timestamp (ISO timestamp; can be null)

iOS

Swift

Request factory method:

```
let request = RestClient.shared.requestForPrimingRecords(relayToken: relayToken,
    changedAfter: changedAfter, apiVersion: nil)
```

Objective-C

Request factory method:

```
- (SFRestRequest*) requestForPrimingRecords:(*nullable* NSString *)relayToken
    changedAfterTimestamp:(nullable NSNumber *)timestamp apiVersion:(*nullable* NSString
 *)apiVersion;
```

Android

Kotlin

Request factory method:

```
val request =
    RestRequest.getRequestForPrimingRecords(ApiVersionStrings.getVersionNumber(this),
        relayToken, changedAfter)
```

Java

Request factory method:

```
public static RestRequest getRequestForPrimingRecords(
    String apiVersion,
    String relayToken,
    Long changedAfterTime)
```

See Also

- "Briefcase Priming Records Resource" in *Connect REST API Developer Guide*
- "Priming Record Collection" in *Connect REST API Developer Guide* (Priming Records response)

Briefcase Priming Records Response

Handles responses for all Mobile SDK Briefcase Priming requests.

In addition to the request factory method, Mobile SDK provides a custom Briefcase response object. Use the properties of this object to obtain parsed Briefcase response values.

Properties

- Priming records (map or dictionary; contains record IDs and their modification timestamps)
- Relay token (string; if null, no more records are available)
- Rule errors (string array; contains IDs of priming rules that were processed but resulted in an error)
- Stats (object containing number of rules, number of records, number of rules served, and number of records served)

iOS

Swift

Response handling:

```
let parsedResponse =
    PrimingRecordsResponse(try response.asJson() as! [AnyHashable : Any])
let records = parsedResponse.primingRecords
let token = parsedResponse.relayToken
let ruleErrors = parsedResponse.ruleErrors
let stats = parsedResponse.stats
```

Objective-C

Response handling:

```
SFSDKPrimingRecordsResponse* parsedResponse = [[SFSDKPrimingRecordsResponse alloc]
    initWith:response];
NSDictionary<NSString*, NSDictionary<NSString*,
    NSArray<SFSDKPrimingRecord*>*>*>* primingRecords =
    parsedResponse.primingRecords;
NSString *relayToken = parsedResponse.relayToken;
NSArray<SFSDKPrimingRuleError*>* ruleErrors = parsedResponse.ruleErrors;
SFSDKPrimingStats * stats = parsedResponse.stats;
```

Android

Kotlin

Request factory method:

```
val parsedResponse = PrimingRecordsResponse(result.asJSONArray())
val primingRecords = parsedResponse.primingRecords
val relayToken = parsedResponse.relayToken
```

```
val ruleErrors = parsedResponse.ruleErrors
val stats = parsedResponse.stats
```

Java

Response handling:

```
PrimingRecordsResponse parsedResponse =
    new PrimingRecordsResponse(result.asJSONObject());
String stats = parsedResponse.stats;
Map<String, Map<String, List<PrimingRecordsResponse.PrimingRecord>>> records =
    parsedResponse.primingRecords;
String relayToken = parsedResponse.relayToken;
List<PrimingRecordsResponse.PrimingRuleError> ruleErrors = parsedResponse.ruleErrors;
```

See Also

- SFSDKPrimingRecordsResponse.h in the `SalesforceMobileSDK-iOS` repo.
- PrimingRecordsResponse.java in the `SalesforceMobileSDK-Android` repo.
- Priming Records request:"Briefcase Priming Records Resource" in *Connect REST API Developer Guide*
- "Priming Record Collection" in *Connect REST API Developer Guide* (Priming Records response)

Collection Create

Creates a collection of records of the specified object type.

Mobile SDK provides a custom response object for parsing Collection request results. See Collection Response.

Parameters

For collections, you can disallow partially successful results by specifying an all-or-none parameter. When you set this parameter to true, Mobile SDK rolls back the entire request if any record creation fails.

- API version (string)
- All or None (Boolean)
- Fields (array)

iOS

Swift

```
let request = RestClient.shared.requestForCreate(allOrNone: allOrNone, records: records,

    apiVersion: nil)
```

Objective-C

```
- (SFRestRequest*) requestForCollectionCreate:(BOOL)allOrNone
                                      records:(NSArray<NSDictionary*>*)records
                                   apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
val request =
    RestRequest.getRequestForCollectionCreate(ApiVersionStrings.getVersionNumber(this),

        allOrNone, records)
```

Java

```
public static RestRequest getRequestForCollectionCreate(String apiVersion,
    boolean allOrNone, JSONArray records)
```

React Native

```
collectionCreate = <T>(
  allOrNone: boolean,
  records: Array<Record<string, unknown>>,
  successCB: ExecSuccessCallback<T>,
  errorCB: ExecErrorCallback,
): void
```

See Also

- "sObject Collections" in *REST API Developer Guide*

Collection Delete

Deletes the objects in a collection that match the given object IDs.

Mobile SDK provides a custom response object for parsing Collection request results. See Collection Response.

Parameters

For collections, you can disallow partial updates by specifying an all-or-none parameter. When you set this parameter to true, Mobile SDK rolls back the entire request if any record deletion fails.

- API version (string)
- All or none (Boolean)
- Object Ids (array)

iOS

Swift

```
let request = RestClient.shared.requestForDelete(allOrNone: allOrNone,
    withObjects: objectIds, apiVersion: nil)
```

Objective-C

```
- (SFRestRequest*)
requestForCollectionDelete:(BOOL)allOrNone
                 objectIds:(NSArray<NSString*>*)objectIds
                apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
val request =
    RestRequest.getRequestForCollectionDelete(ApiVersionStrings.getVersionNumber(this),

        allOrNone, objectIds)
```

Java

```
public static RestRequest getRequestForCollectionDelete(String apiVersion,
    boolean allOrNone, List<String> objectIds)
    throws UnsupportedEncodingException
```

React Native

```
collectionDelete = <T>(
  ids: Array<string>,
  successCB: ExecSuccessCallback<T>,
  errorCB: ExecErrorCallback,
): void
```

See Also

- "SObject Collections" in *REST API Developer Guide*

Collection Retrieve

Retrieves a collection of objects of the given object type that match the given object IDs.

Mobile SDK provides a custom response object for parsing Collection request results. See Collection Response.

Parameters

If you provide a field list, Mobile SDK retrieves only those fields. Otherwise, it returns all accessible standard and custom fields.

- API version (string, optional)
- Object type (string)
- Object IDs (array)
- Field list (array)

iOS

Swift

```
let request = RestClient.shared.requestForRetrieve(withObjectType: objectType,
    objectIds: objectIds!, fieldList: fieldList!, apiVersion: nil)
```

Objective-C

```
- (SFRestRequest*) requestForCollectionRetrieve:(NSString*)objectType
                                      objectIds:(NSArray<NSString*>*)objectIds
                                      fieldList:(NSArray<NSString*>*)fieldList
                                     apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
val request =
   RestRequest.getRequestForCollectionRetrieve(ApiVersionStrings.getVersionNumber(this),

        objectType, objectIds, fieldList)
```

Java

```

RestRequest getRequestForCollectionRetrieve(String apiVersion, String objectType,
    List<String> objectIds, List<String> fieldList)
```

React Native

```
collectionRetrieve = <T>(
  objectType: string,
  ids: Array<string>,
  fields: Array<string>,
  successCB: ExecSuccessCallback<T>,
  errorCB: ExecErrorCallback,
): void
```

See Also

- "sObject Collections" in *REST API Developer Guide*

Collection Update

Updates the requested collection with the given records.

For collections, you can disallow partial updates by specifying an all-or-none parameter. When you set this parameter to true, Mobile SDK rolls back the entire request if any record update fails.

Mobile SDK provides a custom response object for parsing Collection request results. See Collection Response.

Parameters

- API version (string, optional)
- "All or none" preference (Boolean)
- Records (array of sObjects)

iOS

Swift

```
let request = RestClient.shared.requestForCollectionUpdate(allOrNone,
    records:records, apiVersion: apiVersion)
```

Objective-C

```
- (SFRestRequest*) requestForCollectionUpdate:(BOOL)allOrNone
                                      records:(NSArray<NSDictionary*>*)records
                                   apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
val request =
   RestRequest.getRequestForCollectionUpdate(ApiVersionStrings.getVersionNumber(this),

        allOrNone, records)
```

Java

```
public static RestRequest getRequestForCollectionUpdate(String apiVersion,
    boolean allOrNone, JSONArray records) throws JSONException
```

React Native

```
collectionUpdate= <T>(
  allOrNone: boolean,
  records: Array<Record<string, unknown>>,
  successCB: ExecSuccessCallback<T>,
  errorCB: ExecErrorCallback,
): void
```

See Also

- "sObject Collections" in *REST API Developer Guide*

Collection Upsert

Updates or inserts a collection of objects from external data.

Mobile SDK provides a custom response object for parsing Collection request results. See Collection Response.

Parameters

Salesforce inserts or updates a record depending on whether an external ID currently exists in the external ID field. To force Salesforce to create a new record, set the name of the external ID field to "Id" and the external ID value to null.

For collections, you can disallow partial upserts by specifying an all-or-none parameter. When you set this parameter to true, Mobile SDK rolls back the entire request if any record upsert fails.

- API version (string, optional)
- "All or none" preference (Boolean)
- Object type (string)
- External ID field name (string)
- Records (array)

iOS

Swift

```
let request = RestClient.shared.requestForCollectionUpsert(allOrNone: allOrNone,
    withObjectType: objectType, externalIdField: externalIdFieldName!,
    records: records!, apiVersion: nil)
```

Objective-C

```
- (SFRestRequest*)
requestForCollectionUpsert:(BOOL)allOrNone
                objectType:(NSString*)objectType
             externalIdField:(NSString*)externalIdField
                   records:(NSArray<NSDictionary*>*)records
                apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
val request =
   RestRequest.getRequestForCollectionUpsert(ApiVersionStrings.getVersionNumber(this),

        allOrNone, objectType, externalIdFieldName, records)
```

Java

```
public static RestRequest getRequestForCollectionUpsert(
    String apiVersion, boolean allOrNone, String objectType, String externalIdField,
    JSONArray records)
    throws JSONException
```

React Native

```
collectionUpsert = <T>(
  allOrNone: boolean,
  objectType: string,
  externalIdField: string,
```

```
  records: Array<Record<string, unknown>>,
  successCB: ExecSuccessCallback<T>,
  errorCB: ExecErrorCallback,
): void
```

See Also

- "sObject Collections" in *REST API Developer Guide*

Collection Response

Handles responses for all Mobile SDK Collection requests.

Properties:

- Sub-responses (array)
- Sub-response properties:
  - Object ID (string)
  - Success (Boolean)
  - Errors (array)
  - JSON (dictionary)

iOS

Swift, Objective-C

Swift

Response handling:

```
let parsedResponse = CollectionResponse(try response.asJson() as! [Any])
let objId = parsedResponse.subResponses[0].objectId  // String; can be nil
let success = parsedResponse.subResponses[0].success // Boolean
let errors = parsedResponse.subResponses[0].errors   // Array of
CollectionErrorResponse objects with
                                                     // status code, message and
fields
```

Objective-C

Response handling:

```
SFSDKCollectionResponse* parsedResponse = [[SFSDKCollectionResponse alloc]
    initWith:response];
parsedResponse.subResponses[0].objectId
parsedResponse.subResponses[0].success
parsedResponse.subResponses[0].errors // NSArray<SFSDKCollectionErrorResponse*>
with status
                                          // code, message and fields
```

Android

Kotlin

Response handling:

```
val parsedResponse = CollectionResponse(response.asJSONArray())
val objId = parsedResponse.subResponses[0].id         // a string or null
val success = parsedResponse.subResponses[0].success  // a boolean
val errors = parsedResponse.subResponses[0].errors    // a
CollectionSubResponse.ErrorResponse object
                                                      // with status code, message
and fields
```

Java

Response handling:

```
CollectionResponse parsedResponse =
    new CollectionResponse(response.asJSONArray());
String objId = parsedResponse.subResponses.get(0).id; // can be null
Boolean success =
    parsedResponse.subResponses.get(0).success;
List<CollectionResponse.ErrorResponse> errors =
    parsedResponse.subResponses.get(0).errors;
```

See Also

- SFSDKCollectionResponse.h in the `SalesforceMobileSDK-iOS` repo.
- CollectionResponse.java in the `SalesforceMobileSDK-Android` repo.
- "sObject Collections" in *REST API Developer Guide*

Composite Request

Returns a `RestRequest` object that you then use to execute the composite request.

Note: Regardless of the number of subrequests, each composite request counts as one API call.

Parameters

- `apiVersion` (string)
- `requests`
  - **iOS**
    - `requests` (array)—Array of subrequests
    - `refIds` (array)—Array of reference IDs for the requests. The number of elements should match the number of requests.

    Android

    `refIdToRequests` (map)— `LinkedHashMap` of reference IDs to `RestRequest` objects. Requests are played in the order in which they're mapped.
- `allOrNone` (Boolean)—Flag that indicates whether to treat all requests as a single transactional block in error conditions.

iOS

Swift

```
RestClient.shared.compositeRequest(requests:refIds:allOrNone:apiVersion:)
```

Objective-C

```
- (SFRestRequest *)compositeRequest:(NSArray<SFRestRequest *> *) requests
                              refIds:(NSArray<NSString *> *)refIds
                          allOrNone:(BOOL)allOrNone
                         apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
@Throws(JSONException::class)
fun getCompositeRequest(apiVersion: String?, allOrNone: Boolean,
refIdToRequests: LinkedHashMap<String?, RestRequest?>): CompositeRequest
```

Java

```
public static CompositeRequest getCompositeRequest(String apiVersion,
    boolean allOrNone, LinkedHashMap<String, RestRequest> refIdToRequests)
    throws JSONException
```

See Also

- Batch and Composite Requests on page 90 (iOS)
- Batch and Composite Requests on page 141 (Android)
- "Composite" in *REST API Developer Guide*

Create

Creates a record of the specified object type.

Parameters

- API version (string)
- Object type (string)
- (Optional) Map of each field's name (string) to an object containing its value

iOS

Swift

```
RestClient.shared.requestForCreate(withObjectType:fields:)
```

Objective-C

```
- (SFRestRequest *)requestForCreateWithObjectType:(NSString *)objectType
                                           fields:(nullable NSDictionary<NSString*, id> *)fields

                                       apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
fun getRequestForCreate(apiVersion: String?, objectType: String?,
    fields: Map<String?, Any?>?): RestRequest
```

Java

```
public static RestRequest getRequestForCreate(String apiVersion,
    String objectType, Map<String, Object> fields)
```

See Also

- "sObject Basic Information" in *REST API Developer Guide*

Delete

Deletes the object of the given type and the given ID

Parameters

- API version (string)
- Object type (string)
- Object ID (string)

iOS

Swift

```
RestClient.shared.requestForDelete(withObjectType:objectId:)
```

Objective-C

```
- (SFRestRequest *)requestForDeleteWithObjectType:(NSString *)objectType
                                         objectId:(NSString *)objectId
                                       apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
fun getRequestForDelete(apiVersion: String?, objectType: String?, objectId: String?):
RestRequest
```

Java

```
public static RestRequest getRequestForDelete(String apiVersion, String objectType,
String objectId)
```

See Also

- "SObject Rows" in *REST API Developer Guide*

Describe

Completely describes the object's metadata at all levels, including fields, URLs, and child relationships.

Parameters

- API version (string)
- Object type (string)

iOS

Swift

Delegate Method

```
RestClient.shared.requestForDescribe(withObjectType:)
```

Block Method

```
describe(_:onFailure:onSuccess:)
```

Objective-C

Delegate Method

```
- (SFRestRequest *)
    requestForDescribeWithObjectType:(NSString *)objectType
                          apiVersion:(nullable NSString *)apiVersion;
```

Block Method

```
- (SFRestRequest *) performDescribeWithObjectType:(NSString *)objectType
                                    failBlock:(SFRestFailBlock)failBlock

completeBlock:(SFRestDictionaryResponseBlock)completeBlock;
```

Android

Kotlin

```
fun getRequestForDescribe(apiVersion: String?, objectType: String?): RestRequest
```

Java

```
public static RestRequest getRequestForDescribe(String apiVersion, String objectType)
```

See Also

- "sObject Describe" in *REST API Developer Guide*

Describe Global

Returns a list of all available objects in your org and their metadata.

Parameters

- API version (string)

iOS

Swift

```
RestClient.shared.requestForDescribeGlobal()
```

Objective-C

```
- (SFRestRequest *)requestForDescribeGlobal
    apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
fun getRequestForDescribeGlobal(apiVersion: String?): RestRequest
```

Java

```
public static RestRequest getRequestForDescribeGlobal(String apiVersion)
```

See Also

- "Describe Global" in *REST API Developer Guide*

Metadata

Describes metadata provided by sObject basic information for the specified object .

Parameters

- API version (string)
- Object type (string)

iOS

Swift

Delegate Method

```
RestClient.shared.requestForMetadata(withObjectType:apiVersion:)
```

Objective-C

Delegate Method

```
- (SFRestRequest *)
    requestForMetadataWithObjectType:(NSString *)objectType
                          apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
fun getRequestForMetadata(apiVersion: String?, objectType: String?): RestRequest
```

Java

```
public static RestRequest getRequestForMetadata(String apiVersion, String objectType)
```

See Also

- "sObject Basic Information" in *REST API Developer Guide*

Notification

Fetches a single notification by its notification ID.

Parameters

- API version (string)
- Notification ID (string)

iOS

Swift

```
RestClient.shared.request(forNotification:apiVersion:)
```

Objective-C

```
#import <SalesforceSDKCore/SFRestAPI+Notifications.h>
...
- (SFRestRequest *)requestForNotification:(NSString *)notificationId apiVersion:(NSString
 *)apiVersion;
```

Android

Kotlin

```
fun getRequestForNotification(apiVersion: String?, notificationId: String?): RestRequest
```

Java

```
public static RestRequest getRequestForNotification(String apiVersion, String
notificationId)
```

See Also

- "Notification" in *Connect REST API Developer Guide*

Notification Update

Updates the "read" (if non-null) and "seen" (if non-null) statuses of the notification with the given ID.

iOS

In iOS, use the Swift `UpdateNotificationsRequestBuilder` object or the Objective-C `SFSDKUpdateNotificationsRequestBuilder` object to create update requests.

Pass the notification's ID to the `notificationId` property

Swift

```
let builder = UpdateNotificationsRequestBuilder.init()
builder.setNotificationId("<some_id>")
builder.setBefore(Date.init())
builder.setRead(true)
builder.setSeen(true)
let request = builder.buildUpdateNotificationsRequest(SFRestDefaultAPIVersion)
```

Objective-C

```
#import <SalesforceSDKCore/SFRestAPI+Notifications.h>
...

SFSDKUpdateNotificationsRequestBuilder *builder =
    [[SFSDKUpdateNotificationsRequestBuilder alloc] init];
[builder setNotificationId:"<some_id>"]
[builder setRead:true];
[builder setSeen:true];
[builder setBefore: [NSDate date]];
SFRestRequest *updateRequest = [builder
buildUpdateNotificationsRequest:kSFRestDefaultAPIVersion];
```

Android

Parameters

- `apiVersion` (String)

- `notificationId` (String)
- `read` (Boolean)
- `seen` (Boolean)

Kotlin

```
fun getRequestForNotificationUpdate(apiVersion: String?, notificationId: String?,
    read: Boolean?, seen: Boolean?): RestRequest
```

Java

```
public static RestRequest getRequestForNotificationUpdate(String apiVersion,
    String notificationId, Boolean read, Boolean seen)
```

See Also

- "Notification" in *Connect REST API Developer Guide*

Notifications

Gets the given number (maximum 20) of archived Notification Builder notifications based on the given "before" or "after" date.

Parameters

- API version (string)
- Batch size (integer)
- Date before (date, optional)
- Date after (date, optional)

iOS

In iOS, use the `FetchNotificationsRequestBuilder` object or the Objective-C `SFSDKFetchNotificationsRequestBuilder` to create GET requests for notifications.

Swift

```
let builder = FetchNotificationsRequestBuilder.init()
builder.setSize(10)
builder.setBefore(Date.init())
let request = builder.buildFetchNotificationsRequest(SFRestDefaultAPIVersion)
```

Objective-C

```
#import <SalesforceSDKCore/SFRestAPI+Notifications.h>
...

SFSDKFetchNotificationsRequestBuilder *builder =
    [[SFSDKFetchNotificationsRequestBuilder alloc] init];
[builder setBefore: [NSDate date]];
[builder setSize:10];
SFRestRequest *fetchRequest =
    [builder buildFetchNotificationsRequest:kSFRestDefaultAPIVersion];
```

Android

Kotlin

```
fun getRequestForNotifications(apiVersion: String?, size: Int?,
    before: Date?, after: Date?): RestRequest
```

Java

```
public static RestRequest getRequestForNotifications(String apiVersion,
    Integer size, Date before, Date after)
```

See Also

- "Query" in *REST API Developer Guide*

Notifications Status

Get the status of the current user's notifications, including unread and unseen count.

Parameters

- API version (string)

iOS

Swift

For the default `forNotificationsStatus` parameter, pass the API version string.

```
RestClient.shared.request(forNotificationsStatus:)
```

Objective-C

```
#import <SalesforceSDKCore/SFRestAPI+Notifications.h>
...
- (SFRestRequest *)requestForNotificationsStatus:(NSString *)apiVersion;
```

Android

Kotlin

```
fun getRequestForNotificationsStatus(apiVersion: String?): RestRequest
```

Java

```
public static RestRequest getRequestForNotificationsStatus(String apiVersion)
```

See Also

- "Notifications Status" in *Connect REST API Developer Guide*

Notifications Update

Updates the “read” (if non-null) and “seen” (if non-null) statuses of notifications with the given IDs, or those sent before the given date.

iOS

In iOS, use the Swift `UpdateNotificationsRequestBuilder` object or the Objective-C `SFSDKUpdateNotificationsRequestBuilder` object to create update requests.

To define the range of affected notifications, pass either an array of notification IDs or a “before” date. The ID array and “before” date are mutually exclusive parameters.

Swift

```
let builder = UpdateNotificationsRequestBuilder.init()
// builder.setNotificationIds("<array_of_ids>")
// OR
// builder.setBefore(Date.init())
builder.setRead(true)
builder.setSeen(true)
let request = .
    builder.buildUpdateNotificationsRequest(SFRestDefaultAPIVersion)
```

Objective-C

```
#import <SalesforceSDKCore/SFRestAPI+Notifications.h>
...

SFSDKUpdateNotificationsRequestBuilder *builder =
    [[SFSDKUpdateNotificationsRequestBuilder alloc] init];
// [builder setNotificationIds:<array_of_ids>]
// OR
// [builder setBefore: [NSDate date]];
[builder setRead:true];
[builder setSeen:true];
SFRestRequest *updateRequest =
    [builder buildUpdateNotificationsRequest:kSFRestDefaultAPIVersion];
```

Android

Kotlin

```
fun getRequestForNotificationUpdate(apiVersion: String?, notificationId: String?,
    read: Boolean?, seen: Boolean?): RestRequest
```

Java

Parameters

- `apiVersion` (String)
- – `notificationIds` (Array)

  OR

  – `before` (Date)
- `read` (Boolean)

- `seen` (Boolean)

```
public static RestRequest getRequestForNotificationsUpdate(String apiVersion,
    List<String> notificationIds, Date before, Boolean read, Boolean seen)
```

See Also

- "Notifications" in *Connect REST API Developer Guide*

Object Layout

Gets layout metadata for the specified object type and parameters.

Parameters

- API version (string, optional)
- Object API name (string, required)
- Form factor (string, optional)—"Large" (default), "Medium", or "Small"
- Layout type (string, optional)—"Full" (default) or "Compact"
- Mode (string, optional)—"View" (default, "Create", or "Edit"
- Record type ID (string, optional)—The ID of the RecordType object for the new record. If not provided, the default record type is used.

iOS

Swift

```
requestForLayout(withObjectAPIName:formFactor:layoutType:mode:recordTypeId:apiVersion:)
```

Objective-C

```
- (SFRestRequest *)
requestForLayoutWithObjectAPIName:(nonnull NSString *)objectAPIName
                       formFactor:(nullable NSString *)formFactor
                       layoutType:(nullable NSString *)layoutType
                             mode:(nullable NSString *)mode
                     recordTypeId:(nullable NSString *)recordTypeId
                       apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
fun getRequestForObjectLayout(apiVersion: String?, objectAPIName: String?,
    formFactor: String?, layoutType: String?, mode: String?,
    recordTypeId: String?): RestRequest
```

Java

```
public static RestRequest getRequestForObjectLayout(
    String apiVersion, String objectAPIName, String formFactor,
    String layoutType, String mode, String recordTypeId)
```

See Also

- "Get Record Layout Metadata" in *User Interface API Developer Guide*

Resources

Gets available resources for the specified API version, including resource name and URI.

Parameters

- API version (string, optional)

iOS

Swift

```
RestClient.shared.requestForResources()
```

Objective-C

```
- (SFRestRequest *)requestForResources
    apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
fun getRequestForResources(apiVersion: String?): RestRequest
```

Java

```
public static RestRequest getRequestForResources(String apiVersion)
```

See Also

- "Resources by Version" in *REST API Developer Guide*

Retrieve

Retrieves a single sObject record by object ID.

If you provide a list of fields, Mobile SDK retrieves only those fields. Otherwise, it returns all accessible standard and custom fields.

Parameters

- API version (string, optional)
- Object type (string)
- Object ID (string)
- Field list (list of strings, optional)

iOS

In iOS, the `fieldList` parameter expects a comma-separated list of field names, or nil.

Swift

```
RestClient.shared.requestForRetrieve(withObjectType:objectId:fieldList:apiVersion:)
```

Objective-C

```
- (SFRestRequest *)requestForRetrieveWithObjectType:(NSString *)objectType
                                           objectId:(NSString *)objectId
                                          fieldList:(nullable NSString *)fieldList
                                         apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
fun getRequestForRetrieve(apiVersion: String?, objectType: String?,
objectId: String?, fieldList: List<String>?): RestRequest
```

Java

```
public static RestRequest getRequestForRetrieve(String apiVersion, String objectType,
String objectId, List<String> fieldList) throws UnsupportedEncodingException
```

See Also

- For conditions governing field data retrieval, see "Get Field Values from a Standard Object Record" in *REST API Developer Guide*

SOSL Search

Performs the given SOSL search.

Executes the given SOQL query and returns the requested fields of records that satisfy the query.

The `batchSize` parameter can range from 200 to 2,000 (default value) and is not guaranteed to be the actual size at runtime. By default, returns up to 2,000 records at once. If you specify a batch size, this request returns records in batches up to that size. Specifying a batch size does not guarantee that the returned batch is the requested size.

Parameters

- API version (string, optional)
- SOSL query (string)

iOS

Swift

```
RestClient.shared.request(forSearch:apiVersion:)
```

Objective-C

```
- (SFRestRequest *)requestForSearch:(NSString *)sosl
                         apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
@Throws(UnsupportedEncodingException::class)
fun getRequestForSearch(apiVersion: String?, q: String?): RestRequest
```

Java

```
public static RestRequest getRequestForSearch(String apiVersion, String q)
    throws UnsupportedEncodingException
```

See Also

- "Search" in *REST API Developer Guide*

Search Result Layout

Gets the search result layout for up to 100 objects with a single query.

Parameters

- API version (string, optional)
- Object list (list of strings)

iOS

Swift

For the object list, set `forSearchResultLayout` to a string of comma-separated object names.

```
RestClient.shared.request(forSearchResultLayout:apiVersion:)
```

Objective-C

```
- (SFRestRequest *)
requestForSearchResultLayout:(NSString*)objectList
                  apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
@Throws(UnsupportedEncodingException::class)
fun getRequestForSearchResultLayout(apiVersion: String?,
    objectList: List<String>): RestRequest
```

Java

```
public static RestRequest getRequestForSearchResultLayout(String apiVersion, List<String>
 objectList) throws UnsupportedEncodingException
```

See Also

- "Search Result Layouts" in *REST API Developer Guide*

Search Scope and Order

Gets an ordered list of objects in the current user's default global search scope.

Parameters

- API version (string, optional)

iOS

Swift

```
RestClient.shared.requestForSearchScopeAndOrder(apiVersion:)
```

Objective-C

```
- (SFRestRequest *)requestForSearchScopeAndOrder
    apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
fun getRequestForSearchScopeAndOrder(apiVersion: String?): RestRequest
```

Java

```
public static RestRequest getRequestForSearchScopeAndOrder(String apiVersion)
```

See Also

- "Search Scope and Order" in *REST API Developer Guide*

SObject Tree

Creates one or more sObject trees with root records of the specified object type.

Parameters

- API version (string, optional)
- Object type (string)
- Object trees (list or array of sObject tree objects)

iOS

Swift

```
RestClient.shared.request(forSObjectTree:objectTrees:apiVersion:)
```

Objective-C

```
- (SFRestRequest*) requestForSObjectTree:(NSString*)objectType
                                objectTrees:(NSArray<SFSObjectTree*>*)objectTrees
                                 apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
@Throws(JSONException::class)
fun getRequestForSObjectTree(apiVersion: String?, objectType: String?,
    objectTrees: List<SObjectTree>): RestRequest
```

Java

```
public static RestRequest getRequestForSObjectTree(String apiVersion, String objectType,
 List<SObjectTree> objectTrees) throws JSONException
```

See Also

- "sObject Tree" in *REST API Developer Guide*

SOQL Query

Executes the given SOQL query and returns the requested fields of records that satisfy the query.

The `batchSize` parameter can range from 200 to 2,000 (default value) and is not guaranteed to be the actual size at runtime. By default, returns up to 2,000 records at once. If you specify a batch size, this request returns records in batches up to that size. Specifying a batch size does not guarantee that the returned batch is the requested size.

Parameters

- API version (string, optional)
- Query (string)

- Batch size (integer)

iOS

Swift

```
RestClient.shared.request(forQuery:apiVersion:batchSize:)
```

Objective-C

```
- (SFRestRequest *)requestForQuery:(NSString *)soql
                        apiVersion:(nullable NSString *)apiVersion
                         batchSize:(NSInteger)batchSize;
```

Android

Kotlin

```
@Throws(UnsupportedEncodingException::class)
fun getRequestForQuery(apiVersion: String?, q: String?): RestRequest
```

Java

```
public static RestRequest
    getRequestForQuery(String apiVersion, String q)
    throws UnsupportedEncodingException

public static RestRequest
    getRequestForQuery(String apiVersion, String q, int batchSize)
    throws UnsupportedEncodingException
```

See Also

- "Query" in *REST API Developer Guide*

SOQL Query All

Executes the given SOQL string. The result includes all current and deleted objects that satisfy the query.

The `batchSize` parameter can range from 200 to 2,000 (default value) and is not guaranteed to be the actual size at runtime. By default, returns up to 2,000 records at once. If you specify a batch size, this request returns records in batches up to that size. Specifying a batch size does not guarantee that the returned batch is the requested size.

Parameters

- API version (string, optional)
- Query (string)

iOS

Swift

```
RestClient.shared.request(forQueryAll:apiVersion:)
```

Objective-C

```
- (SFRestRequest *)requestForQueryAll:(NSString *)soql
                           apiVersion:(nullable NSString *)apiVersion;
```

Android

Not supported.

See Also

- "QueryAll" in *REST API Developer Guide*

User Info

Returns information associated with the current user.

iOS

Swift

```
RestClient.shared.requestForUserInfo()
```

Objective-C

```
- (SFRestRequest *)requestForUserInfo;
```

Android

Kotlin

```
val requestForUserInfo: RestRequest
```

Java

```
public static RestRequest getRequestForUserInfo()
```

See Also

- "Query for User Information" in *Salesforce Help*

Update

Updates specified fields of the requested record with the given values. Can also prevent the update from occurring if the record has been modified since a given date.

Parameters

- API version (string, optional)
- Object type (string)
- Object ID (string)
- Fields (map, optional)—Maps fields to be updated to their new values
- "If unmodified since" date (date, optional)—Fulfills the request only if the record hasn't been modified since the given date

iOS

Swift

```
RestClient.shared.requestForUpdate(withObjectType:objectId:fields:)

RestClient.shared.requestForUpdate(withObjectType:objectId:fields:ifUnmodifiedSince:)
```

Objective-C

```
- (SFRestRequest *)requestForUpdateWithObjectType:(NSString *)objectType
                                         objectId:(NSString *)objectId
                                           fields:(nullable NSDictionary<NSString*, id> *)fields

                                       apiVersion:(nullable NSString *)apiVersion;

- (SFRestRequest *)requestForUpdateWithObjectType:(NSString *)objectType
                                         objectId:(NSString *)objectId
                                           fields:(nullable NSDictionary<NSString*, id> *)fields

                             ifUnmodifiedSinceDate:(nullable NSDate *) ifUnmodifiedSinceDate
                                       apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
fun getRequestForUpdate(apiVersion: String?, objectType: String?,
    objectId: String?, fields: Map<String?, Any?>?): RestRequest

fun getRequestForUpdate(apiVersion: String?, objectType: String?,
    objectId: String?, fields: Map<String?, Any?>?,
    ifUnmodifiedSinceDate: Date?): RestRequest
```

Java

```
public static RestRequest getRequestForUpdate(String apiVersion,
    String objectType, String objectId, Map<String, Object> fields)

public static RestRequest getRequestForUpdate(String apiVersion,
    String objectType, String objectId, Map<String, Object> fields,
    Date ifUnmodifiedSinceDate)
```

See Also

- "Update a Record" in *REST API Developer Guide*

Upsert

Updates or inserts an object from external data.

Salesforce inserts or updates a record depending on whether the external ID currently exists in the external ID field. To force Salesforce to create a new record, set the name of the external ID field to "Id" and the external ID to null.

Parameters

- API version (string, optional)
- Object type (string)
- External ID field (string)
- External ID (string, optional)
- Fields (map, optional)—Maps each field name to an object containing its value

If `fields` is null, the upserted record is empty.

iOS

Swift

```
RestClient.shared.requestForUpsert(withObjectType:externalIdField:externalId:fields:)
```

Objective-C

```
- (SFRestRequest *)requestForUpsertWithObjectType:(NSString *)objectType
                                  externalIdField:(NSString *)externalIdField
                                       externalId:(nullable NSString *)externalId
                                           fields:(NSDictionary<NSString*, id> *)fields
                                       apiVersion:(nullable NSString *)apiVersion;
```

Android

Kotlin

```
fun getRequestForUpsert(apiVersion: String?, objectType: String?,
    externalIdField: String?, externalId: String?,
    fields: Map<String?, Any?>?): RestRequest
```

Java

```
public static RestRequest getRequestForUpsert(String apiVersion,
    String objectType, String externalIdField, String externalId,
    Map<String, Object> fields)
```

See Also

- "Insert or Update (Upsert) a Record Using an External ID" in *REST API Developer Guide*

Versions

Gets summary information about each Salesforce API version currently available.

iOS

Swift

```
RestClient.shared.requestForVersions()
```

Objective-C

```
- (SFRestRequest *)requestForVersions;
```

Android

Kotlin

```
val requestForVersions: RestRequest
```

Java

```
public static RestRequest getRequestForVersions()
```

See Also

- "Versions" in *REST API Developer Guide*

Files API Reference

For information about using the Files APIs, see Files and Networking. Here are links to the API references:

Android
See Package com.salesforce.androidsdk.rest.files.

iOS
See "Files Methods" at
https://forcedotcom.github.io/SalesforceMobileSDK-iOS/Documentation/SalesforceSDKCore/html/Classes/SFRestAPI.html.

Hybrid
Hybrid file request wrappers reside in the `force+files.js` JavaScript library.

iOS Architecture

Mobile SDK is essentially a single library that includes the following modules:

- `SalesforceSDKCommon`—Implements utilities and custom data types used across other iOS components.
- `SalesforceAnalytics`—Implements a logging and instrumentation framework for Mobile SDK.
- `SalesforceSDKCore`—Implements OAuth authentication, passcode, and networking.
- `SmartStore`—Mobile SDK offline secure storage solution.
- `MobileSync`—Mobile SDK offline synchronization solution.

The following iOS architectures depend on the iOS native library but are maintained in separate GitHub repos:

- React Native—Native bridges to Mobile SDK features. For use only in React Native apps.
- Hybrid—Defines the Mobile SDK Cordova plugin for Mobile SDK features. For use only in hybrid apps.

If you use forceios to create native apps, CocoaPods incorporates the required modules based on the app type you specify. If you create native apps with a clone of the SalesforceMobileSDK-iOS git repo, your project uses these modules as dynamic libraries.

SEE ALSO:

Logging and Analytics

Supported REST Services

Mobile SDK REST APIs support the standard object operations offered by Salesforce Platform REST and SOAP APIs. For most operation types, a factory method or factory object creates a REST request object specifically for that operation. You send this request object to Salesforce using the Mobile SDK REST API and receive the response asynchronously.

After you've sent a request to Salesforce, the response arrives in your app asynchronously. To handle these responses, you can specify a callback delegate when you send the request, or define a closure (Swift only).

Notification

Get a notification.

Delegate Method

Swift

```
RestClient.shared.request(forNotification:apiVersion:)
```

Objective-C

```
- (SFRestRequest *)requestForNotification:(NSString *)notificationId apiVersion:(NSString
 *)apiVersion;
```

Block Method

Not available.

Notifications Status

Get the status of a range of notifications, including unread and unseen count.

Delegate Method

Swift

```
RestClient.shared.request(forNotificationsStatus:)
```

Objective-C

```
- (SFRestRequest *)requestForNotificationsStatus:(NSString *)apiVersion;
```

Block Method

Not available.

Notifications

Get the given number (maximum 20) of archived Notification Builder notifications based on the given "before" or "after" date. In Mobile SDK, use the Swift `FetchNotificationsRequestBuilder` object or the Objective-C `SFSDKFetchNotificationsRequestBuilder` to create GET requests for notifications.

Delegate Method

Swift

```
let builder = FetchNotificationsRequestBuilder.init()
builder.setSize(10)
builder.setBefore(Date.init())
let request = builder.buildFetchNotificationsRequest(SFRestDefaultAPIVersion)
```

Objective-C

```
SFSDKFetchNotificationsRequestBuilder *builder =
    [[SFSDKFetchNotificationsRequestBuilder alloc] init];
[builder setBefore: [NSDate date]];
[builder setSize:10];
SFRestRequest *fetchRequest =
    [builder buildFetchNotificationsRequest:kSFRestDefaultAPIVersion];
```

Block Method

Not available.

Notifications Update

Update the "read" and "seen" statuses of a given set of Notification Builder notifications. In Mobile SDK, use the Swift `UpdateNotificationsRequestBuilder` object or the Objective-C `SFSDKUpdateNotificationsRequestBuilder` object to create update requests.

To update a single notification, set the `notificationId` property. To update a range of notifications, set either the `notificationIds` or the `before` property. These properties—`notificationId`, `notificationIds`, and `before`—are mutually exclusive.

Delegate Method

Swift

```
let builder = UpdateNotificationsRequestBuilder.init()
builder.setBefore(Date.init())
builder.setRead(true)
builder.setSeen(true)
let request = builder.buildUpdateNotificationsRequest(SFRestDefaultAPIVersion)
```

Objective-C

```
SFSDKUpdateNotificationsRequestBuilder *builder =
    [[SFSDKUpdateNotificationsRequestBuilder alloc] init];
[builder setRead:true];
[builder setSeen:true];
[builder setBefore: [NSDate date]];
SFRestRequest *updateRequest = [builder
buildUpdateNotificationsRequest:kSFRestDefaultAPIVersion];
```

Block Method

Not available.

Example: For sample calls, see `/libs/SalesforceSDKCore/SalesforceSDKCoreTests/SalesforceRestAPITests.m` at github.com/forcedotcom/SalesforceMobileSDK-iOS.

SEE ALSO:

Supported Salesforce APIs

Android Architecture

Salesforce Mobile SDK for Android is a library project that includes Mobile SDK core components, SmartStore, and Mobile Sync. It also incorporates hybrid and React Native native bridges for Android.

Android apps reference the `SalesforceSDK` project from their application project. See the Android developer documentation.

Android Packages and Classes

Java source files for the Android Mobile SDK are under `libs/SalesforceSDK/src`.

Catalog of Top-Level Packages

For package and class descriptions, see the Salesforce Mobile SDK Android Reference.

Android Resources

In Mobile SDK projects, resources are under the `/res` folder. In the SalesforceMobileSDK-Android GitHub repo, you can find them at libs/SalesforceSDK/res.

Supported Versions of Tools and Components for Mobile SDK 11.1

All Platforms

Tool or Component	Supported Version	Installation Details
Node.js	Latest	Install from nodejs.org
npm	3.10	Installed by Node.js
shelljs	0.8.5	Installed by Node.js
SQLite	3.41.2	Installed by Mobile SDK
SQLCipher	4.5.4 for Android, 4.5.4 for iOS	Installed by Mobile SDK
Full Text Search (FTS)	FTS5	Installed by Mobile SDK

iOS

Tool or Component	Supported Version	Installation Details
Xcode	14	Install from the Mac App Store
iOS Deployment Target	15	Installed by Xcode
iOS Base SDK	16	Installed by Xcode
CocoaPods	1.8 to no declared maximum	Install from cocoapods.org
forceios	11.1	At a command line or Terminal prompt, type: `npm install -g forceios`

Android

Tool or Component	Supported Version	Installation Details
Java JDK	11.0.11+9	Install from oracle.com
Android Studio	Latest	Install from developer.android.com/studio/
Gradle	7.2.1	Installed by Android Studio
Android SDK minApi	Android Nougat (API 24)	Install through the Android SDK Manager in Android Studio
Android SDK targetApi	Android 13 (API 33)	Install through the Android SDK Manager in Android Studio
Default Android SDK version for hybrid apps	Target version is Android 13 (API 33), minimum version is Android Nougat (API 24)	Install through the Android SDK Manager in Android Studio
OkHttp	3.12.1	Installed by Mobile SDK
forcedroid	11.1	At a command line or Terminal prompt, type: `npm install -g forcedroid`

Hybrid

Tool or Component	Supported Version	Installation Details
Cordova	12.0.1 (for Android), 7.0.1 (for iOS)	Install from cordova.apache.org
Cordova command line	12.0.0	At a command line or Terminal prompt, type: `npm install -g cordova`

Tool or Component	Supported Version	Installation Details
Default Android SDK version for hybrid apps	Target version is Android 13 (API 33), minimum version is Android Nougat (API 24)	Install through the Android SDK Manager in Android Studio

React Native

Tool or Component	Supported Version	Installation Details
React Native	0.70.14	Installed by Mobile SDK
React	18.1.0	Installed by Mobile SDK
forcereact	11.1	At a command line or Terminal prompt, type: `npm install -g forcereact`

CHAPTER 25 iOS Current Deprecations

These lists show currently deprecated Mobile SDK objects and artifacts for iOS, as annotated in the source files. Use this information to prepare for the removal of these artifacts in the release indicated.

SFApplication

```
@interface SFApplication : UIApplication
```

- Deprecated in 11.1 for removal in 12.0.

SFInactivityTimerCenter

```
@interface SFInactivityTimerCenter : NSObject
```

- Deprecated in 11.1 for removal in 12.0.

CHAPTER 26 iOS APIs Removed in Mobile SDK 11.0

These lists show Mobile SDK objects and artifacts for iOS that were removed in Mobile SDK 10.0.

SFCryptChunks

```
@interface SFCryptChunks : NSObject
```

SFDecryptStream

```
@interface SFDecryptStream : NSInputStream <SFCryptChunksDelegate>
```

- Use `SFSDKDecryptStream` instead. This class should only be used for upgrade steps.

SFEncryptionKey

```
@interface SFEncryptionKey : NSObject <NSCoding, NSCopying>
```

SFKeyStoreManager

```
@interface SFKeyStoreManager : NSObject
```

SFSecureEncryptionKey

```
@interface SFSecureEncryptionKey : SFEncryptionKey
```

SFSoupSpec

```
@interface SFSoupSpec : NSObject
```

- External storage and soup spec have been removed in 11.0.

SFAlterSoupLongOperation

```
- (id) initWithStore:(SFSmartStore*)store
           soupName:(NSString*)soupName
       newSoupSpec:(nullable SFSoupSpec*)newSoupSpec
```

```
        newIndexSpecs:(NSArray*)newIndexSpecs
          reIndexData:(BOOL)reIndexData;
```

- External storage and soup spec have been removed in 11.0 - use other constructor instead.

SFSmartStore

```
- (nullable SFSoupSpec*)attributesForSoup:(NSString*)soupName
NS_SWIFT_NAME(specification(forSoupNamed:));
```

- External storage and soup spec have been removed in 11.0.

```
- (BOOL)registerSoupWithSpec:(SFSoupSpec*)soupSpec
               withIndexSpecs:(NSArray<SFSoupIndex*>*)indexSpecs
                        error:(NSError**)error
NS_SWIFT_NAME(registerSoup(withSpecification:withIndices:));
```

- External storage and soup spec have been removed in 11.0. Use `registerSoup` with `soupName` instead.

```
- (unsigned long
long)getExternalFileStorageSizeForSoup:(NSString*)soupName
NS_SWIFT_NAME(externalFileStorageSize(forSoupNamed:));
```

- External storage and soup spec have been removed in 11.0.

```
- (NSUInteger)getExternalFilesCountForSoup:(NSString*)soupName
NS_SWIFT_NAME(externalFilesCount(forSoupNamed:));
```

- External storage and soup spec have been removed in 11.0.

```
- (BOOL) alterSoup:(NSString*)soupName
      withSoupSpec:(SFSoupSpec*)soupSpec
    withIndexSpecs:(NSArray<SFSoupIndex*>*)indexSpecs
       reIndexData:(BOOL)reIndexData
NS_SWIFT_NAME(alterSoup(named:soupSpec:indexSpecs:reIndexData:));
```

- External storage and soup spec have been removed in 11.0. Use other `alterSoup` method instead.

CHAPTER 27 Android Current Deprecations

These lists show currently deprecated Mobile SDK objects and artifacts for Android, as annotated in the source files. Use this information to prepare for the removal of these artifacts in the release indicated.

Mobile SDK for Android hasn't tagged any APIs for deprecation in the 11.1 release.

However, be prepared for the following changes planned for Mobile SDK 12.0.

Upgrade to Gradle 8

The upgrade to Gradle 8 in Mobile SDK 12.0 will require you to upgrade your build files to Gradle 8 and use JDK 17.

Upgrade Firebase Cloud Messaging Beyond 20.1.0

Mobile SDK 12.0 will upgrade FCM to the latest version after 20.1.0. To upgrade your apps, complete these steps.

1. Remove the `androidPushNotificationClientId` bootconfig entry.
2. Add `google-services.json` to the root of your project.

CHAPTER 28 Android APIs Removed in Mobile SDK 11.0

These lists show Mobile SDK objects and artifacts for Android that were removed in Mobile SDK 10.0.

Mobile SDK 10.0 for Android has removed the following APIs.

SoupSpec

```
public class SoupSpec
```

- External storage and soup spec have been removed in 11.0.

AlterSoupLongOperation

```
public AlterSoupLongOperation(SmartStore store, String soupName,

    SoupSpec newSoupSpec, IndexSpec[] newIndexSpecs, boolean
reIndexData)
    throws JSONException
```

- External storage and soup spec have been removed in 11.0. Use other constructor instead.

SmartStore

```
public void registerSoupWithSpec(final SoupSpec soupSpec,
    final IndexSpec[] indexSpecs)
```

- External storage and soup spec have been removed in 11.0. Use `registerSoup(String soupName, IndexSpec[] indexSpecs)` instead.

```
public void alterSoup(String soupName, SoupSpec soupSpec,
    IndexSpec[] indexSpecs, boolean reIndexData)
    throws JSONException
```

- External storage and soup spec have been removed in 11.0. Use `alterSoup(String soupName, IndexSpec[] indexSpecs)` instead.

```
public SoupSpec getSoupSpec(String soupName)
```

- External storage and soup spec have been removed in 11.0

```
public boolean usesExternalStorage(String soupName)
```

- External storage and soup spec have been removed in 11.0

CHAPTER 29 Trademark Attributions

App Store, iOS, iTunes, MacBook, Objective-C, OS X, Swift, TV, Watch, and Xcode are trademarks of Apple Inc.

Google, Google+, Chrome, Play, and Android are trademarks of Alphabet Inc.

Microsoft, Windows, Visual Studio, Access, and Azure are trademarks of Microsoft, Inc.

Amazon is a trademark of Amazon.com, Inc.

Facebook is a trademark of Facebook, Inc.

PayPal is a trademark of PayPal Holdings, Inc.

LinkedIn is a trademark of LinkedIn, Inc.

Twitter is a trademark of Twitter, Inc.

CHAPTER 30 Removing UIWebView from iOS Hybrid Apps

In 2018, Apple deprecated `UIWebView` in favor of `WKWebView`. The App Store recently announced a timeline for formally removing apps that still use `UIWebView`. To conform to these requirements, Mobile SDK has removed all its references to `UIWebView`.

Mobile SDK phased in the `UIWebView` replacement over three release cycles. However, when Mobile SDK 8.0 was being developed, Cordova—the underlying technology for Mobile SDK hybrid apps—still referenced `UIWebView` in its code base. With no means of removing these unused references, Mobile SDK 8.0 couldn't satisfy the new App Store requirements.

Mobile SDK 8.1 adopts the new Cordova 5.1.1 release, which allows clients to "compile out" its `UIWebView` code. As a result, Mobile SDK 8.1 is ready for App Store submittals.

Frequently Asked Questions

What are the App Store deadlines?

- **April 2020:** App Store stops accepting new iOS apps that contain references to `UIWebView`.
- **December 2020:** App Store stops accepting updated iOS apps that contain references to `UIWebView`.

Which apps are affected?

This change affects Mobile SDK hybrid apps on iOS. It does not affect Mobile SDK hybrid apps on Android, Mobile SDK native apps, or Mobile SDK React Native apps.

What actions do I need to take?

- To submit **new** hybrid apps to the App Store: Upgrade to Mobile SDK 8.1 by April 2020.
- To submit **updated** hybrid apps to the App Store: Upgrade to Mobile SDK 8.1 (or later) by December 2020.
- For **all** hybrid apps:
  - Update any `UIWebView` references in custom code to `WKWebView`.
  - While updating to 8.1, apply code changes as needed where other APIs have been deprecated, removed, or replaced.

What is UIWebView?

`UIWebView` is a deprecated iOS user interface control in Apple's UIKit framework. It loads HTML files and web content into an app view, rendering them as they would appear in a browser window. See developer.apple.com/documentation/uikit/uiwebview.

See Also

Apple Developer News Release

INDEX

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W