



Version 1: 1

Mobile App Developer Guide



Last updated: July 4, 2014

Table of Contents

Preface.....	1
Salesforce Platform Mobile Services.....	2
Mobile Services in Force.com.....	2
Salesforce Mobile SDK.....	3
Identity.....	3
Salesforce1 Platform.....	3
When to Use Salesforce1 Platform vs. Creating a Custom App.....	4
About This Book.....	4
Version.....	5
Sending Feedback.....	5
Chapter 1: Introduction to Mobile Development.....	6
About Native, HTML5, and Hybrid Development.....	7
Multi-Device Strategy.....	9
Developer Edition or Sandbox Environment?.....	12
Development Prerequisites.....	13
Sign Up for Force.com.....	14
Supported Browsers.....	14
Enough Talk; I'm Ready.....	16
Chapter 2: Getting Started.....	17
Creating a Connected App.....	18
Create a Connected App.....	18
Installing Mobile SDK.....	20
Mobile SDK npm Packages.....	20
Do This First: Install Node.js and npm.....	20
iOS Installation.....	21
Android Installation.....	21
Uninstalling Mobile SDK npm Packages.....	22
Mobile SDK GitHub Repository.....	23
Mobile SDK Sample Apps.....	23
Installing the Sample Apps.....	24
Installing Sample Apps for Android.....	24
Installing Sample Apps for iOS.....	25
What's New.....	26
What's New in Mobile SDK 2.0.....	27
What's New in Mobile SDK 2.1.....	28
HTML5 Improvements in Visualforce (Winter '14 Release).....	28
Chapter 3: HTML5 and Hybrid Development.....	30
Getting Started.....	31
Using HTML5 and JavaScript.....	31
HTML5 Development Requirements.....	31

HTML5 Development Tools.....	31
Mobile Design Templates.....	31
HTML5 Mobile Templates Sample App.....	32
Using Mobile Design Templates in Visualforce.....	33
Data Binding with Mobile Templates.....	34
Using JavaScript Remoting to Query Contact Records.....	34
Using Underscore to Generate the Template Markup.....	35
Customizing Look and Feel.....	36
List View Templates.....	37
Detail View Templates.....	45
Data Input Templates.....	49
Map View Templates.....	56
Calendar View Templates.....	58
Report and Dashboard Templates.....	61
Miscellaneous Templates.....	64
Mobile Packs.....	66
jQuery Quick Start.....	67
Angular.js Quick Start.....	68
Backbone.js Quick Start.....	70
Knockout Quick Start.....	71
Mobile UI Elements.....	72
Using the Camera in HTML5: Mobile UI Elements Sample App.....	75
Delivering HTML5 Content With Visualforce.....	80
Accessing Salesforce Data: Controllers vs. APIs.....	80
Introduction to Hybrid Development.....	82
iOS Hybrid Development.....	83
Android Hybrid Development.....	83
JavaScript Files for Hybrid Applications.....	83
Hybrid Apps Quick Start.....	84
Running the Sample Hybrid App.....	85
How the Sample App Works.....	89
Create a Mobile Page to List Information.....	90
Create a Mobile Page for Detailed Information.....	93
Guidelines and Tips for Hybrid Apps.....	95
Versioning and Javascript Library Compatibility.....	95
Example: Serving the Appropriate Javascript Libraries.....	97
Managing Sessions in Hybrid Applications.....	98
Chapter 4: Native iOS Development.....	101
iOS Native Quick Start.....	102
Native iOS Requirements.....	102
Creating an iOS Project.....	102
Running the Xcode Project Template App.....	104
Developing a Native iOS App.....	104
About Login and Passcodes.....	105

About Memory Management.....	105
Overview of Application Flow.....	105
AppDelegate Class.....	106
About View Controllers.....	107
RootViewController Class.....	108
About Salesforce REST APIs.....	109
Supported Operations.....	109
SFRestAPI Interface.....	111
SFRestDelegate Protocol.....	111
Creating REST Requests.....	113
Sending a REST Request.....	113
SFRestRequest Class.....	114
Using SFRestRequest Methods.....	114
SFRestAPI (Blocks) Category.....	115
SFRestAPI (QueryBuilder) Category.....	116
SFRestAPI (Files) Category.....	118
Tutorial: Creating a Native iOS Warehouse App.....	119
Create a Native iOS App.....	120
Step 1: Create a Connected App.....	120
Step 2: Create a Native iOS Project.....	121
Step 3: Run the New iOS App.....	122
Step 4: Explore How the iOS App Works.....	123
Customize the List Screen.....	125
Step 1: Modify the Root View Controller.....	125
Step 2: Create the App's Root View	126
Step 3: Try Out the App.....	126
Create the Detail Screen.....	127
Step 1: Create the App's Detail View Controller.....	127
Step 2: Set Up DetailViewController.....	129
Step 3: Create the Designated Initializer.....	131
Step 4: Establish Communication Between the View Controllers.....	133
Step 5: Try Out the App.....	139
iOS Native Sample Applications.....	139
Chapter 5: Native Android Development.....	140
Android Native Quick Start.....	141
Native Android Requirements.....	141
Creating an Android Project.....	141
Setting Up Sample Projects in Eclipse.....	144
Android Project Files.....	144
Developing a Native Android App.....	145
The create_native Script.....	145
Android Application Structure.....	145
Native API Packages.....	147
Overview of Native Classes.....	148

SalesforceSDKManager Class.....	148
KeyInterface Interface.....	149
AccountWatcher Class.....	149
PasscodeManager Class.....	150
Encryptor class.....	151
SalesforceActivity, SalesforceListActivity, and SalesforceExpandableListActivity Classes.....	151
UI Classes.....	151
ClientManager Class.....	151
RestClient Class.....	152
RestRequest Class.....	152
FileRequests Class.....	153
WrappedRestRequest Class.....	155
LoginActivity Class.....	155
Other UI Classes.....	155
UpgradeManager Class.....	155
Utility Classes.....	155
ForcePlugin Class.....	156
Using Passcodes.....	156
Resource Handling.....	157
Using REST APIs.....	159
Android Template App: Deep Dive.....	162
TemplateApp Class.....	162
MainActivity Class.....	163
TemplateApp Manifest.....	164
Tutorial: Creating a Native Android Warehouse Application.....	164
Prerequisites.....	164
Create a Native Android App.....	166
Step 1: Create a Connected App.....	166
Step 2: Create a Native Android Project.....	166
Step 3: Run the New Android App.....	167
Step 4: Explore How the Android App Works.....	167
Customize the List Screen.....	168
Step 1: Remove Existing Controls.....	168
Step 2: Update the SOQL Query.....	169
Step 3: Try Out the App.....	171
Create the Detail Screen.....	171
Step 1: Create the Detail Screen.....	171
Step 2: Create the DetailActivity Class.....	173
Step 3: Customize the DetailActivity Class.....	173
Step 4: Link the Two Activities, Part 1: Create a Data Class.....	174
Step 5: Link the Two Activities, Part 2: Implement a List Item Click Handler.....	175
Step 6: Implement the Update Button.....	177
Step 7: Try Out the App.....	179
Android Native Sample Applications.....	179

Chapter 6: Files and Networking.....	180
Architecture.....	181
Downloading Files and Managing Sharing.....	181
Uploading Files.....	181
Encryption and Caching.....	182
Using Files in Android Apps.....	182
Managing the Request Queue.....	182
Using Files in iOS Native Apps.....	183
Managing Requests.....	184
Using Files in Hybrid Apps.....	185
Chapter 7: Offline Management.....	186
Securely Storing Data Offline.....	187
About SmartStore.....	187
SmartStore Soups.....	187
SmartStore Data Types.....	187
Enabling SmartStore in Hybrid Apps.....	188
Adding SmartStore to Existing Android Apps.....	188
Registering a Soup.....	189
Retrieving Data From a Soup.....	190
Smart SQL Queries.....	193
Working With Cursors.....	194
Manipulating Data.....	195
Using the Mock SmartStore.....	197
NativeSqlAggregator Sample App: Using SmartStore in Native Apps.....	198
Using SmartSync to Access Salesforce Objects.....	200
About Backbone Technology.....	200
Models and Model Collections.....	201
Models.....	201
Model Collections.....	202
Using the SmartSync Data Framework in JavaScript.....	203
Offline Caching.....	204
Implementing Offline Caching.....	206
Using StoreCache For Offline Caching.....	207
Conflict Detection.....	210
Mini-Tutorial: Conflict Detection.....	212
Accessing Custom API Endpoints.....	214
Force.RemoteObject Class.....	214
Force.RemoteObjectCollection Class.....	214
Using Apex REST Resources.....	216
Using External Objects (Beta).....	219
Tutorial: Creating a SmartSync Application.....	220
Set Up Your Project.....	220
Edit the Application HTML File.....	221

Create a SmartSync Model and a Collection.....	223
Create a Template.....	224
Add the Search View.....	225
Add the Search Result List View.....	226
Add the Search Result List Item View.....	227
Router.....	228
SmartSync Sample Apps.....	232
User and Group Search Sample.....	235
User Search Sample.....	237
Account Editor Sample.....	240
Chapter 8: Push Notifications and Mobile SDK.....	248
About Push Notifications.....	249
Using Push Notifications in Android.....	249
Configure a Connected App For GCM (Android).....	249
Code Modifications (Android).....	250
Using Push Notifications in iOS.....	251
Configure a Connected App for APNS (iOS).....	251
Code Modifications (iOS).....	252
Chapter 9: Using Communities With Mobile SDK Apps.....	255
Communities and Mobile SDK Apps.....	256
Set Up an API-Enabled Profile.....	256
Set Up a Permission Set.....	257
Grant API Access to Users.....	258
Configure the Login Endpoint.....	258
Branding Your Community.....	259
Customizing Communities Login.....	260
Using External Authentication With Communities.....	262
About External Authentication Providers.....	262
Using the Community URL Parameter.....	263
Using the Scope Parameter.....	264
Configuring a Facebook Authentication Provider.....	265
Configuring a Salesforce Authentication Provider.....	267
Configuring an OpenID Connect Authentication Provider.....	269
Example: Configure a Community For Mobile SDK App Access.....	271
Add Permissions to a Profile.....	271
Create a Community.....	272
Add the API User Profile To Your Community.....	272
Create a New Contact and User.....	272
Test Your New Community Login.....	273
Example: Configure a Community For Facebook Authentication.....	274
Create a Facebook App.....	274
Define a Salesforce Auth. Provider.....	274
Configure Your Facebook App.....	275

Customize the Auth. Provider Apex Class.....	276
Configure Your Salesforce Community.....	276
Chapter 10: Authentication, Security, and Identity in Mobile Apps.....	278
OAuth Terminology.....	279
OAuth2 Authentication Flow.....	279
OAuth 2.0 User-Agent Flow.....	280
OAuth 2.0 Refresh Token Flow.....	281
Scope Parameter Values.....	281
Using Identity URLs.....	282
Setting a Custom Login Server.....	287
Revoking OAuth Tokens.....	288
Handling Refresh Token Revocation in Android Native Apps.....	288
Token Revocation Events.....	288
Token Revocation: Passive Handling.....	289
Token Revocation: Active Handling.....	289
Connected Apps.....	290
About PIN Security.....	290
Portal Authentication Using OAuth 2.0 and Force.com Sites.....	291
Chapter 11: Distributing Mobile AppExchange Apps.....	292
AppExchange for Mobile: Enterprise Mobile Apps.....	293
Joining the AppExchange Partner Program.....	293
Get a Publishing Org.....	294
Create a Provider Profile.....	294
The AppExchange Security Review.....	294
Index.....	296

Preface

Mobile devices have radically changed the way we work and play. People consume, create, and share data on a wide range of connected devices. Workers use smart phones and tablets to stay in touch, connect with customers and peers, and engage on social networks and apps.

However, many companies continue to run their businesses on enterprise applications that don't work in the mobile world. These legacy applications remain locked away on corporate intranets and aren't available in employees' hands when they're needed. They don't provide a modern user experience, and they aren't wired into social graphs like consumer apps.

Yesterday's platforms were not designed to meet the demands of the mobile world. Big, monolithic stacks and rigid integration patterns lack the scalability and flexibility required by mobile technology. Techniques that evolved since the 1990s for web applications on PCs don't apply in mobile apps. Mobile applications require new architectures and software designs, and they need to run on platforms built for mobile application development and wireless connectivity. Today, outdated corporate applications are rapidly being replaced by mobile-ready cloud apps.

Table 1: Comparison of PC/Web applications and a modern mobile application

Category	Typical PC / Web application	Mobile / modern application
Connection and Availability	<ul style="list-style-type: none">• Fast, reliable LAN• Low latency• High bandwidth• Connectivity assumed	<ul style="list-style-type: none">• Varying connection• High latency• Low bandwidth• Offline operation required
User Interactions	<ul style="list-style-type: none">• Keyboard and mouse• Long desktop interactions	<ul style="list-style-type: none">• Touch screen• Quick, focused actions
Perimeter Security	<ul style="list-style-type: none">• Corporate VPN or LAN access to applications	<ul style="list-style-type: none">• Cumbersome to require VPN from mobile devices• IP restrictions ineffective with public mobile networks
Device Standardization	<ul style="list-style-type: none">• Typically purchased and controlled by IT	<ul style="list-style-type: none">• Often Bring Your Own Device (BYOD)• Multiple platforms
Form Factor	<ul style="list-style-type: none">• Large (PC) screen	<ul style="list-style-type: none">• Apps must support phone, tablet, and desktop
Social	<ul style="list-style-type: none">• Typically siloed applications• Email-based collaboration	<ul style="list-style-type: none">• Native user collaboration• Intuitively share and collaborate
Multi-device	<ul style="list-style-type: none">• Client-server architectures with data stored on server (Web)	<ul style="list-style-type: none">• Instant sharing between devices• Data propagation between devices

Category	Typical PC / Web application	Mobile / modern application
Device Interaction	<ul style="list-style-type: none"> Applications rarely leverage telephony, camera, and other media devices 	<ul style="list-style-type: none"> Native use of mobile device's camera, contacts, calendar, and location
Location	<ul style="list-style-type: none"> Rarely used in Web applications 	<ul style="list-style-type: none"> Commonly used both to associate data with a location and to filter data and services based on location

Salesforce provides a state-of-the-art cloud-based platform for building CRM mobile apps. The Salesforce Mobile SDK gives you advanced control over mobile device features, offline support, data synchronization, and mobile software design.

Salesforce Platform Mobile Services

Enterprise IT departments now face the daunting task of connecting their enterprise data and services with a mobile workforce. Salesforce faced this problem itself as it moved its enterprise CRM and service applications to the mobile world. This transformation required fundamental changes in the underlying technology and implementation to support Salesforce's applications across multiple platforms (iOS, Android) and multiple form factors (phone, tablet, and PC) with enterprise-grade reliability, availability, and security. The lessons learned and technology built to transform Salesforce's applications for mobile are now available for any company that uses the Salesforce cloud.

Salesforce Platform Mobile Services are designed to meet the challenges of mobile applications.

Salesforce Platform Mobile Services is the next-generation platform that powers Salesforce mobile applications, enabling enterprises to build their own Android, iPhone, and iPad applications. These services leverage the power of the Salesforce platform and its proven security, reliability, and scale for enterprise applications.

Salesforce Platform Mobile Services comprises three core components.

- Mobile Services in Force.com
- Salesforce Mobile SDK 2.1
- Identity

Mobile Services in Force.com

Mobile services in Force.com focus on developing and administering enterprise mobile applications.

- Mobile REST APIs provide access to enterprise data and services, leveraging standard Web protocols. Developers can quickly access their business data through REST APIs and leverage that data across phone, tablet, and web user interfaces. The REST APIs provide a single place to enforce access, security, common policy across all device types.
- Social (Chatter) REST APIs enable developers to quickly transform their applications with social networks and collaboration features. The Chatter REST API provides access to the feed, as well as the social graph of user connections. Mobile applications can easily consume or post items to a user or group, or leverage the social graph to enable instant collaboration between connected users.
- Mobile policy management enables administrators to enforce their enterprise security policy on mobile applications in a world without perimeter security. Administrators can enable security features such as two-factor authentication, device PIN protection, and password rotation. They can also enable and disable user access to mobile applications.
- Geolocation provides location-based information to enhance your online business processes with geospatial data. All objects in Salesforce include a compound geolocation field. The entire platform is location-ready, allowing radius-based searching and other spatial queries.

Salesforce Mobile SDK

Salesforce Mobile SDK lets you develop native Objective-C apps for iOS and Java apps for Android. You can also use it to provide a native container for hybrid apps written in HTML5 and JavaScript. Npm scripts for iOS and Android help you get started building native and hybrid apps. Salesforce Mobile SDK provides:

- Native device services. You can access device features such as the camera, GPS, and contacts across a broad range of iOS and Android devices.
- Secure offline storage and data synchronization. You can build applications which continue to function with limited or no network connectivity. The data stored on the device is securely encrypted and safe, even if the device is lost or stolen.
- Client OAuth authentication support. You're free from having to rebuild login pages and general authentication in mobile apps. Mobile SDK apps quickly and easily integrate with enterprise security management.

Identity

Identity provides a single enterprise identity and sign-on service to connect mobile devices with enterprise data and services. Identity provides the following advantages.

- Single sign-on across applications and devices, so users aren't forced to create multiple usernames and passwords.
- A trusted identity provider that you can leverage for any enterprise platform or application.
- A Cloud Directory that enables enterprises to white label identity services and use company-specific appearance and branding.
- The ability to utilize consumer identity providers, such as Facebook. This feature allows customer-facing applications to quickly engage with customer social data.

Salesforce1 Platform

If you attended Dreamforce '13 or have browsed developer.salesforce.com/docs, you've probably heard about the Salesforce1 Platform. Salesforce Mobile SDK is part of this platform, but Salesforce1 Platform also provides other alternatives for developing mobile apps. You might be asking yourself, "What exactly is Salesforce1 Platform?"

The Salesforce1 Platform is designed to deliver a customer benefit behind every app. It's API- and mobile-first, where every aspect can be extended and customized by every user, regardless of whether they work within lines of business, in IT, or are looking to build an entire company and product. And every app is instantly mobile.

Salesforce1 as an engineering philosophy means:

- Every new feature must be designed for mobile first and have an API for developers.
- User interfaces should be responsive and change dynamically depending on whether the app is running on a smartphone, tablet, or laptop.
- The user experience should change depending on device features.
- Address fields should leverage geolocation and provide maps, nearby information, and context. You should be able to click on a phone number to make a call.
- Apps should be personal. Your identity should drive the user interface by interacting with calendars, personal preferences, and usage history.
- The entire platform should grow with your needs, constantly delivering customer benefit for every app and every action.

Salesforce1 is a transformation, not a reinvention. If you're an existing customer with data, applications, custom logic, or user interfaces built on Salesforce, this investment is now instantly mobile-aware.

A full discussion of Salesforce1 Platform development is beyond the scope of this book, but you can learn more at developer.salesforce.com/docs.

When to Use Salesforce1 Platform vs. Creating a Custom App

When it comes to developing functionality for your Salesforce mobile users, you have options. Although this book deals only with Mobile SDK development, Salesforce also provides the Salesforce1 Platform for mobile app development.

Here are some differences between extending Salesforce1 and creating custom apps using the Mobile SDK. For more information on Salesforce1, see developer.salesforce.com/docs.

Salesforce1 Platform

- Has a defined user interface.
- Has full access to Salesforce data.
- You can create an integrated experience with functionality developed in the Salesforce1 Platform.
- Publisher actions give you a way to include your own apps/functionality.
- You can customize Salesforce1 with point-and-click or programmatic customizations.
- Functionality can be added programmatically through Visualforce pages or Force.com Canvas apps.
- Salesforce1 customizations or apps adhere to the Salesforce1 navigation. So, for example, a Visualforce page can be called from the navigation menu or from the publisher.
- You can leverage existing Salesforce development experience, both point-and-click and programmatic.
- Included in all Salesforce editions and supported by salesforce.com.

Custom Apps

Custom apps can be either free-standing apps you create with Salesforce Mobile SDK or browser apps using plain HTML5 and JQuery Mobile/Ajax. With custom apps, you can:

- Define a custom user experience.
- Access Salesforce data using REST APIs in native and hybrid local apps, or with Visualforce in hybrid apps using JavaScript Remoting. In HTML5 apps, do the same using JQueryMobile and Ajax.
- Brand your user interface for customer-facing exposure.
- Create standalone mobile apps, either with native APIs using Java for Android or Objective-C for iOS, or through a hybrid container using JavaScript and HTML5 (Mobile SDK only).
- Distribute apps through mobile industry channels, such as the Apple App Store or Google Play (Mobile SDK only).
- Configure and control complex offline behavior (Mobile SDK only).
- Use push notifications (Developer Preview in Winter '14; available for Mobile SDK native apps only).
- 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. No licensing required.
 - ◇ Requires you to develop and compile your apps in an external development environment (Xcode for iOS, Eclipse or similar for Android).
 - ◇ Development costs range from \$0 to \$1M or more, plus maintenance costs.

About This Book

This book introduces you to Salesforce Platform Mobile Services and teaches you how to design, develop, and manage mobile applications for the cloud. The chapters cover a wide range of development techniques for various skill sets, beginning with HTML5 and JavaScript, continuing through hybrid apps, and culminating in native iOS and Android development.

Each development paradigm is represented by a quick start tutorial. Most of these tutorials take you through the steps of creating a simple master-detail application that accesses Salesforce through REST APIs. Tutorials include:

- [HTML5 Mobile Design Templates](#) on page 31
- [Hybrid Apps Quick Start](#) on page 84
- [Tutorial: Creating a Native Android Warehouse Application](#) on page 164
- [Tutorial: Creating a Native iOS Warehouse App](#) on page 119
- [Tutorial: Creating a SmartSync Application](#) on page 220

You'll also find pointers to Mobile SDK sample apps, tips and techniques for working with communities and managing hybrid apps, and descriptions of Mobile SDK "feature" APIs such as:

- [Files and Networking](#) on page 180
- [Securely Storing Data Offline](#) on page 187
- [Using SmartSync to Access Salesforce Objects](#) on page 200
- [Push Notifications and Mobile SDK](#) on page 248

Enjoy your journey through Salesforce Platform Mobile Services!



Note: An online version of this book is available at developer.salesforce.com/docs.

Version

This book was last revised on October 24th, 2013, and was verified to work with the Salesforce Winter '14 release and Mobile SDK version 2.1.

Sending Feedback

Questions or comments about anything you see in this book? Suggestions for topics that you'd like to see covered in future versions? Go to the Force.com discussion boards at <http://boards.developerforce.com> and let us know what you think! Or email us directly at developerforce@salesforce.com.

Chapter 1

Introduction to Mobile Development

In this chapter ...

- [About Native, HTML5, and Hybrid Development](#)
- [Multi-Device Strategy](#)
- [Developer Edition or Sandbox Environment?](#)
- [Development Prerequisites](#)
- [Supported Browsers](#)
- [Enough Talk; I'm Ready](#)

Force.com has proven itself as an easy and highly productive platform for cloud computing. Developers can define application components, such as custom objects and fields, workflow rules, Visualforce pages, and Apex classes and triggers, and assemble them into killer apps.

The Mobile SDK seamlessly integrates with the Force.com cloud architecture and provides:

- SmartSync Data Framework for accessing Salesforce data through JavaScript
- Secure offline storage
- Data syncing for hybrid apps
- Implementation of Force.com Connected App policy that works out of the box
- OAuth credentials management, including persistence and refresh capabilities
- Wrappers for Salesforce REST APIs
- Libraries for building native iOS and Android applications
- Containers for building hybrid applications



Note:

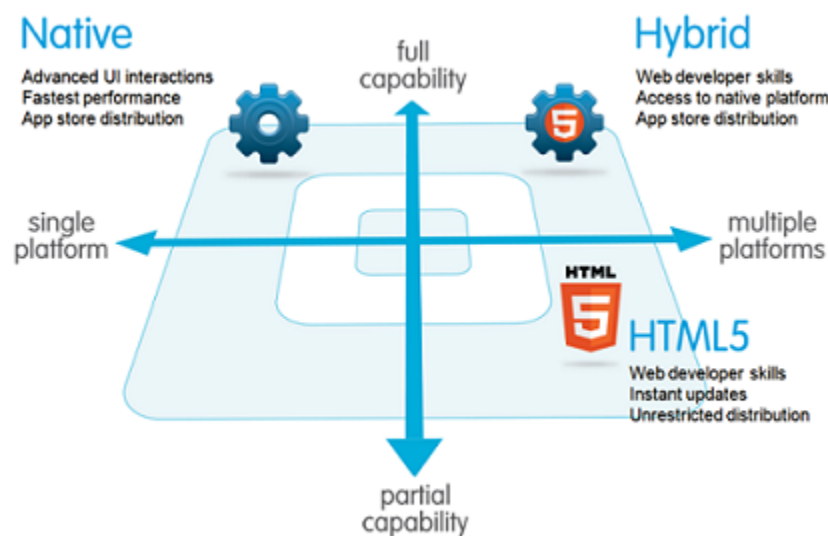
Be sure to visit [Salesforce Platform Mobile Services](#) website regularly for tutorials, blog postings, and other updates.

About Native, HTML5, and Hybrid Development

Salesforce Mobile SDK gives you options for how you'll develop your app. The option you choose depends on your development skills, device and technology requirements, goals, and schedule.

The Mobile SDK offers three ways to create mobile apps:

- **Native** apps are specific to a given mobile platform (iOS or Android) and use the development tools and language that the respective platform supports (for example, Xcode and Objective-C with iOS, Eclipse and Java with Android). Native apps look and perform best but require the most development effort.
- **HTML5** apps use standard web technologies—typically HTML5, JavaScript and CSS—to deliver apps through a mobile Web browser. This “write once, run anywhere” approach to mobile development creates cross-platform mobile applications that work on multiple devices. While developers can create sophisticated apps with HTML5 and JavaScript alone, some challenges remain, such as session management, secure offline storage, and access to native device functionality (such as camera, calendar, notifications, and so on).
- **Hybrid** apps combine the ease of HTML5 Web app development with the power of the native platform by wrapping a Web app inside the Salesforce container. This combined approach produces an application that can leverage the device's native capabilities and be delivered through the app store. You can also create hybrid apps using Visualforce pages delivered through the Salesforce hybrid container.



Native Apps

Native apps provide the best usability, the best features, and the best overall mobile experience. There are some things you get only with native apps:

- **Fast graphics API**—the native platform gives you the fastest graphics, which might not be a big deal if you're showing a static screen with only a few elements, or a very big deal if you're using a lot of data and require a fast refresh.
- **Fluid animation**—related to the fast graphics API is the ability to have fluid animation. This is especially important in gaming, highly interactive reporting, or intensely computational algorithms for transforming photos and sounds.
- **Built-in components**—The camera, address book, geolocation, and other features native to the device can be seamlessly integrated into mobile apps. Another important built-in component is encrypted storage, but more about that later.

- **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 apps are usually developed using an integrated development environment (IDE). IDEs provide tools for building, debugging, project management, version control, and other tools professional developers need. You need these tools because native apps are more difficult to develop. Likewise, the level of experience required is higher than in other development scenarios. If you're a professional developer, you don't have to be sold on proven APIs and frameworks, painless special effects through established components, or the benefits of having all your code in one place.

HTML5 Apps

An HTML5 mobile app is basically a web page, or series of web pages, that are designed to work on a small mobile device screen. As such, HTML5 apps are device agnostic and can be opened with any modern mobile browser. Because your content is on the web, it's searchable, which can be a huge benefit for certain types of apps (shopping, for example).

If you're new to mobile development, the technological bar is lower for Web apps; it's easier to get started here than in native or hybrid development. Unfortunately, every mobile device seems to have its own idea of what constitutes usable screen size and resolution. This diversity imposes an additional burden of testing on different devices. Browser incompatibility is especially common on Android devices, for example.

An important part of the "write once, run anywhere" HTML5 methodology is that distribution and support is much easier than for native apps. Need to make a bug fix or add features? Done and deployed for all users. For a native app, there are longer development and testing cycles, after which the consumer typically must log into a store and download a new version to get the latest fix.

If HTML5 apps are easier to develop, easier to support, and can reach the widest range of devices, where do these apps lose out?

- **Secure offline storage**—HTML5 browsers support offline databases and caching, but with no out-of-the-box encryption support. You get all three features in Mobile SDK native applications.
- **Security**—In general, implementing even trivial security measures on a native platform can be complex tasks for a mobile Web developer. It can also be painful for users. For example, a web app with authentication requires users to enter their credentials every time the app restarts or returns from a background state.
- **Native features**—The camera, address book, and other native features are accessible on limited, if any, browser platforms.
- **Native look and feel**—HTML5 can only emulate the native look, while customers won't be able to use familiar compound gestures.

Hybrid Apps

Hybrid apps are built using HTML5 and JavaScript wrapped inside a thin container that provides access to native platform features. For the most part, hybrid apps provide the best of both worlds, being almost as easy to develop as HTML5 apps with all the functionality of native. In addition, hybrid apps can use the SmartSync Data Framework in JavaScript to

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

You know that native apps are installed on the device, while HTML5 apps reside on a Web server, so you might be wondering whether hybrid apps store their files on the device or on a server? You can implement a hybrid app locally or remotely.

Locally

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

Remotely

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 browser window.

Both types of hybrid development are covered in this guide.

Native, HTML5, and Hybrid Summary

The following table sums up how the three mobile development scenarios stack up.

	Native	HTML5	Hybrid
Graphics	Native APIs	HTML, Canvas, SVG	HTML, Canvas, SVG
Performance	Fastest	Fast	Fast
Look and feel	Native	Emulated	Emulated
Distribution	App store	Web	App store
Camera	Yes	Browser dependent	Yes
Notifications	Yes	No	Yes
Contacts, calendar	Yes	No	Yes
Offline storage	Secure file system	Not secure; shared SQL, Key-Value stores	Secure file system; shared SQL
Geolocation	Yes	Yes	Yes
Swipe	Yes	Yes	Yes
Pinch, spread	Yes	Yes	Yes
Connectivity	Online, offline	Mostly online	Online, offline
Development skills	Objective C, Java	HTML5, CSS, JavaScript	HTML5, CSS, JavaScript

Multi-Device Strategy

With the proliferation of mobile devices in this post-PC era, applications now have to support a variety of platforms, form factors, and device capabilities. Some of the key considerations and design options for Force.com developers looking to develop such device-independent applications are:

- Which devices and form factors should your app support?
- How does your app detect various types of devices?
- How should you design a Force.com 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) will play 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). touch.salesforce.com is another example of a user experience that is customized for a specific form factor. 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): <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 and the Device.js library will take care of automatically redirecting 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;
    }
}
```

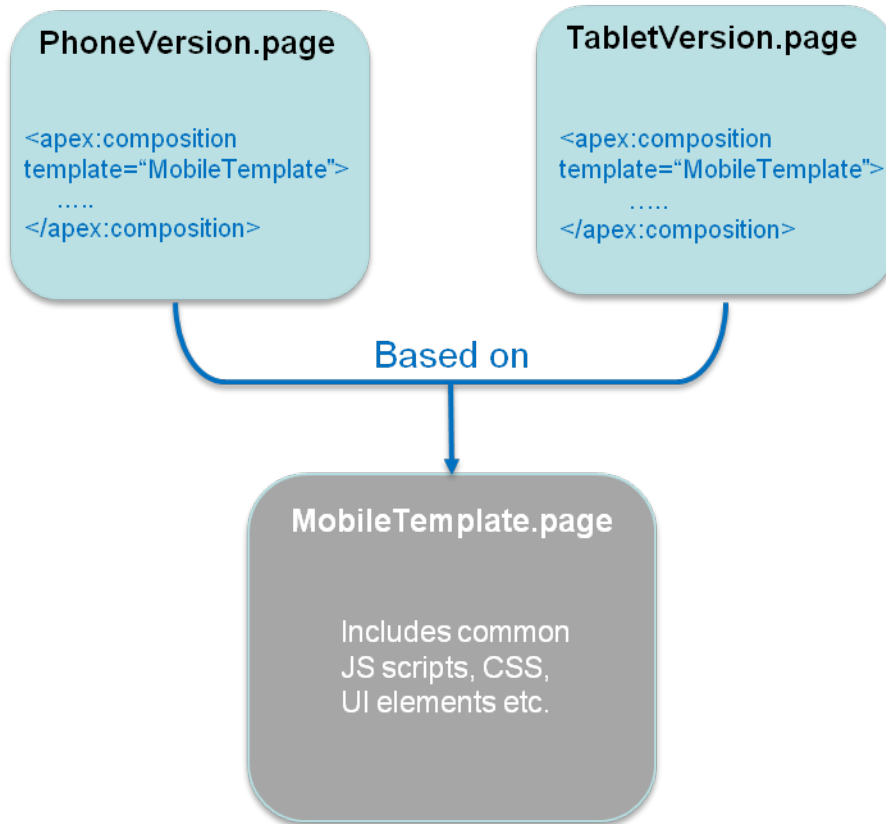
Note that User-Agent parsing in the code snippet above is far from comprehensive and you should implement something more robust that detects all the devices that you need to support based on regular expression matching. A good place to start is to look at the RegEx included in the detectmobilebrowsers.com code snippets.

How Should You Design a Force.com 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 <div> tag) to selectively show/hide page elements. This approach however can result in bloated and hard-to-maintain code and should be used sparingly.

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 are not business critical. Development can be done inside your browser or with the Force.com IDE, which is based on the Eclipse development tool. There are two types of development environments: Developer Edition and Sandbox.

Types of Developer Environments

A *Developer Edition* (DE) environment is a free, fully-featured copy of the Enterprise Edition environment, with less storage and users. DE is a logically separate environment, ideal as your initial development environment. You can sign-up for as many DE organizations 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 DE 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 Enterprise or Unlimited Edition customers. The sandbox copy can include data, configurations, or both. It is possible to 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 (DE) organization. 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 are a partner who intends to build a commercially available Force.com app by creating a managed package for distribution through AppExchange and/or Trialforce.



Note: Only Developer Edition or Partner Developer Edition environments can create managed packages.

- You are a salesforce.com customer with a Professional, Group, or Personal Edition, and you do not have access to Sandbox.
- You are a developer looking to explore the Force.com platform for FREE!

Partner Developer Edition is ideal if:

- You are developing in a team and you require a master environment to manage all the source code - in this case each developer would have a Developer Edition environment and check their code in and out of this master repository environment.
- You expect more than 2 developers to log in to develop and test.
- You require a larger environment that allows more users to run robust tests against larger data sets.

Sandbox is ideal if:

- You are a salesforce.com customer with Enterprise, Unlimited, or Force.com Edition, which includes Sandbox.
- You are developing a Force.com application specifically for your production environment.
- You are not planning to build a Force.com application that will be distributed commercially.
- You have no intent to list on the AppExchange or distribute through Trialforce.

Development Prerequisites

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

It's helpful to have some experience with Force.com. You'll need a Force.com Developer Edition organization.

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](#).

The following requirements apply to specific platforms and technologies:

- To build iOS applications (hybrid or native), see [Native iOS Requirements](#).
- To build Android applications (hybrid or native), see [Native Android Requirements](#).
- To build remote hybrid applications, you'll need an organization that has Visualforce.

Sign Up for Force.com

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

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 `Master Subscription 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.

Supported Browsers

Learn about the browsers we support for the full Salesforce site.



Important: Beginning Summer '15, we'll discontinue support for Microsoft® Internet Explorer® versions 7 and 8. For these versions, this means that some functions may no longer work after this date. Salesforce.com Customer Support will not investigate issues related to Internet Explorer 7 and 8 after this date.

To see the mobile browsers that are supported for the Salesforce1 app, check out “Requirements for Using the Salesforce1 App” in the Salesforce Help.

Browser	Comments
Microsoft® Internet Explorer® versions 7, 8, 9, 10, and 11	<p>If you use Internet Explorer, we recommend using the latest version that Salesforce supports. Apply all Microsoft software updates. Note these restrictions.</p> <ul style="list-style-type: none"> • The full Salesforce site is not supported in Internet Explorer on touch-enabled devices for Windows. Use the Salesforce1 mobile browser app instead. • The HTML solution editor in Internet Explorer 11 is not supported in Salesforce Knowledge. • The Compatibility View feature in Internet Explorer isn't supported. • The Metro version of Internet Explorer 10 isn't supported. • Internet Explorer 6 and 7 aren't supported for login hints for multiple accounts. • Internet Explorer 7 and 8 aren't supported for the Developer Console or the Data Import Wizard. • Internet Explorer 7 isn't supported for Open CTI. • Internet Explorer 7 and 11 aren't supported for Salesforce CRM Call Center built with CTI Toolkit version 4.0 or higher. • Internet Explorer 7 isn't supported for Force.com Canvas.

Browser	Comments
	<ul style="list-style-type: none"> Internet Explorer 7 isn't supported for Salesforce console features that require more advanced browser performance and recent Web technologies. The console features not available in Internet Explorer 7 include: <ul style="list-style-type: none"> ◊ The Most Recent Tabs component ◊ Multiple custom console components on sidebars ◊ Multi-monitor components ◊ The resizable highlights panel ◊ The full-width feed option on feed-based page layouts Internet Explorer 7 and 8 aren't supported for Community Templates for Self-Service. Community Templates for Self-Service supports Internet Explorer 9 and above for desktop users and Internet Explorer 11 and above for mobile users. <p>For configuration recommendations, see "Configuring Internet Explorer" in the Salesforce Help.</p>
Mozilla® Firefox®, most recent stable version	<p>Salesforce.com makes every effort to test and support the most recent version of Firefox.</p> <ul style="list-style-type: none"> Mozilla Firefox is supported for desktop users only for Community Templates for Self-Service. <p>For configuration recommendations, see "Configuring Firefox" in the Salesforce Help.</p>
Google Chrome™, most recent stable version	<p>Google Chrome applies updates automatically; salesforce.com makes every effort to test and support the most recent version. There are no configuration recommendations for Chrome. Chrome isn't supported for the Console tab or the Add Google Doc to Salesforce browser button.</p>
Apple® Safari® versions 5.x and 6.x on Mac OS X	<p>There are no configuration recommendations for Safari. Apple Safari on iOS isn't supported for the full Salesforce site.</p> <ul style="list-style-type: none"> Safari isn't supported for the Salesforce console. Safari isn't supported for Salesforce CRM Call Center built with CTI Toolkit versions below 4.0.

Recommendations and Requirements for All Browsers

- For all browsers, you must enable JavaScript, cookies, and SSL 3.0.
- Salesforce.com recommends a minimum screen resolution of 1024 x 768 for the best possible user experience. Screen resolutions smaller than 1024 x 768 may not display Salesforce features such as Report Builder and Page Layout Editor properly.
- For Mac OS users on Apple Safari or Google Chrome, make sure the system setting `Show scroll bars` is set to **Always**.
- Some third-party Web browser plug-ins and extensions can interfere with the functionality of Chatter. If you experience malfunctions or inconsistent behavior with Chatter, disable all of the Web browser's plug-ins and extensions and try again.

Certain features in Salesforce—as well as some desktop clients, toolkits, and adapters—have their own browser requirements. For example:

- Internet Explorer is the only supported browser for:

- ◇ Standard mail merge
- ◇ Installing Salesforce Classic on a Windows Mobile device
- ◇ Connect Offline
- Firefox is recommended for the enhanced page layout editor.
- Browser requirements also apply for uploading multiple files on Chatter.

Discontinued or Limited Browser Support

As of Summer '12, salesforce.com discontinued support for Microsoft® Internet Explorer® 6. Existing features that have previously worked in this browser may continue to work through 2014. Note these support restrictions.

- Internet Explorer 6 isn't supported for:
 - ◇ Chatter
 - ◇ Global search
 - ◇ Answers
 - ◇ Cloud Scheduler
 - ◇ The new user interface theme
 - ◇ Quote Template Editor
 - ◇ Salesforce console
 - ◇ Salesforce Knowledge
 - ◇ Live Agent
 - ◇ Forecasts
 - ◇ Chatter Answers
 - ◇ Enhanced profile user interface
 - ◇ Site.com
 - ◇ Schema Builder
 - ◇ Joined reports
 - ◇ Enhanced dashboard charting options

Internet Explorer 7 isn't supported for Site.com and Chatter Messenger. For systems running Microsoft Windows XP, Internet Explorer versions 7 and 8 with the latest security patches are supported for Chatter Answers.

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](#) on page 84
- [iOS Native Quick Start](#) on page 102
- [Android Native Quick Start](#) on page 141

Chapter 2

Getting Started

In this chapter ...

- [Creating a Connected App](#)
- [Installing Mobile SDK](#)
- [Mobile SDK Sample Apps](#)
- [What's New](#)

Let's get started creating custom mobile apps! You need a Connected App definition regardless of which development options you choose. For every development path except plain HTML5 browser apps, install the Salesforce Mobile SDK. If you plan to convert a plain HTML5 browser app into a standalone hybrid app, you'll also need Salesforce Mobile SDK.

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

To create a connected app, you use the Salesforce app.

1. Log into your Force.com instance.
2. In Setup, navigate to **Create > Apps**.
3. Under Connected Apps, click **New**.
4. Perform steps for [Basic Information](#).
5. Perform steps for [API \(Enable OAuth Settings\)](#).
6. Click **Save**.



Note:

- The `Callback URL` provided for OAuth does not 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://`.
- 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.

See also [Scope Parameter Values](#).

Basic Information

Specify basic information about your app in this section, including the app name, logo, and contact information.

1. Enter the `Connected App Name`. This name is displayed in the list of connected apps.



Note: The name must be unique for the current connected apps in your organization. You can reuse the name of a deleted connected app if the connected app was created using the Spring '14 release or later. You cannot reuse the name of a deleted connected app if the connected app was created using an earlier release.

2. Enter the `API Name`, used when referring to your app from a program. It defaults to a version of the name without spaces. Only letters, numbers, and underscores are allowed, so you'll need to edit the default name if the original app name contained any other characters.
3. Provide the `Contact Email` that salesforce.com should use for contacting you or your support team. This address is not provided to administrators installing the app.
4. Provide the `Contact Phone` for salesforce.com to use in case we need to contact you. This number is not provided to administrators installing the app.
5. Enter a `Logo Image URL` to display your logo in the list of connected apps and on the consent page that users see when authenticating. The URL must use HTTPS. The logo image can't be larger than 125 pixels high or 200 pixels wide, and must be in the GIF, JPG, or PNG file format with a 100 KB maximum file size. The default logo is a cloud. You have several ways to add a custom logo.
 - You can upload your own logo image by clicking **Upload logo image**. Select an image from your local file system that meets the size requirements for the logo. When your upload is successful, the URL to the logo appears in the `Logo Image URL` field. Otherwise, make sure the logo meets the size requirements.

- You can also select a logo from the samples provided by clicking **Choose one of our sample logos**. The logos available include ones for Salesforce apps, third-party apps, and standards bodies. Click the logo you want, and then copy and paste the displayed URL into the `Logo Image URL` field.
 - You can use a logo hosted publicly on Salesforce servers by uploading an image that meets the logo file requirements (125 pixels high or 200 pixels wide, maximum, and in the GIF, JPG, or PNG file format with a 100 KB maximum file size) as a document using the Documents tab. Then, view the image to get the URL, and enter the URL into the `Logo Image URL` field.
- Enter an `Icon URL` to display a logo on the OAuth approval page that users see when they first use your app. The logo should be 16 pixels high and wide, on a white background. Sample logos are also available for icons.

You can select an icon from the samples provided by clicking **Choose one of our sample logos**. Click the icon you want, and then copy and paste the displayed URL into the `Icon URL` field.
 - If there is a Web page with more information about your app, provide a `Info URL`.
 - Enter a `Description` to be displayed in the list of connected apps.

Prior to Winter '14, the `Start URL` and `Mobile Start URL` were defined in this section. These fields can now be found under Web App Settings and Mobile App Settings below.

API (Enable OAuth Settings)

This section controls how your app communicates with Salesforce. Select `Enable OAuth Settings` to configure authentication settings.

- Enter the `Callback URL` (endpoint) that Salesforce calls back to your application during OAuth; it's the `OAuth redirect_uri`. Depending on which OAuth flow you use, this is typically the URL that a user's browser is redirected to after successful authentication. As this URL is used for some OAuth flows to pass an access token, the URL must use secure HTTP (HTTPS) or a custom URI scheme.
- If you're using the JWT OAuth flow, select `Use Digital Signatures`. If the app uses a certificate, click **Choose File** and select the certificate file.
- Add all supported OAuth scopes to `Selected OAuth Scopes`. These scopes refer to permissions given by the user running the connected app, and are followed by their OAuth token name in parentheses:

Access and manage your Chatter feed (`chatter_api`)

Allows access to Chatter REST API resources only.

Access and manage your data (`api`)

Allows access to the logged-in user's account using APIs, such as REST API and Bulk API. This value also includes `chatter_api`, which allows access to Chatter REST API resources.

Access your basic information (`id, profile, email, address, phone`)

Allows access to the Identity URL service.

Access custom permissions (`custom_permissions`)

Allows access to the custom permissions in an organization associated with the connected app, and shows whether the current user has each permission enabled.



Note: Custom permissions are currently available as a Developer Preview.

Allow access to your unique identifier (`openid`)

Allows access to the logged in user's unique identifier for OpenID Connect apps.

Full access (full)

Allows access to all data accessible by the logged-in user, and encompasses all other scopes. `full` does not return a refresh token. You must explicitly request the `refresh_token` scope to get a refresh token.

Perform requests on your behalf at any time (refresh_token, offline_access)

Allows a refresh token to be returned if you are eligible to receive one. This lets the app interact with the user's data while the user is offline. The `refresh_token` scope is synonymous with `offline_access`.

Provide access to custom applications (visualforce)

Allows access to Visualforce pages.

Provide access to your data via the Web (web)

Allows the ability to use the `access_token` on the Web. This also includes `visualforce`, allowing access to Visualforce pages.

If your organization had the `No user approval required for users in this organization` option selected on your remote access prior to the Spring '12 release, users in the same organization as the one the app was created in still have automatic approval for the app. The read-only `No user approval required for users in this organization` checkbox is selected to show this condition. For connected apps, the recommended procedure after you've created an app is for administrators to install the app and then set `Permitted Users to Admin-approved users`. If the remote access option was not checked originally, the checkbox doesn't display.

Installing Mobile SDK

Salesforce Mobile SDK provides two installation paths.

- (*Recommended*) You can install the SDK in a ready-made development setup using a Node Packaged Module (npm) script.
- You can download the Mobile SDK open source code from GitHub and set up your own development environment.

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 two npm packages: **forceios** for iOS, and **forcedroid** for Android.

Mobile SDK npm packages 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 install Mobile SDK, but also to create new template projects and install the SDK samples.

Npm packages for the Salesforce Mobile SDK reside at <https://www.npmjs.org>.



Note: 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](https://www.npmjs.org) website and download the new package.

Do This First: Install Node.js and npm

To use the Mobile SDK npm installers, install Node.js. The Node.js installer automatically installs npm.

1. Download Node.js from www.nodejs.org/download.
2. Run the downloaded installer to install Node.js and npm. Accept all prompts that ask for permission to install.

3. Test your installation at a command prompt by typing `npm`, then pressing ENTER or RETURN. If you don't see a page of command usage information, revisit Step 2 to find out what's missing.

Now you're ready to download the npm scripts and install Salesforce Mobile SDK for Android and iOS.

iOS Installation

For the fastest, easiest route to iOS development, use the `forceios` npm package to install Salesforce Mobile SDK.

1. At a command prompt, use the `forceios` package to install the Mobile SDK either globally (recommended) or locally.
 - a. **For global installation:** Use the `sudo` command and append the "global" option, `-g`:

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

With the `-g` option, you can run `npm install` from any directory. The npm utility installs the package under `/usr/local/lib/node_modules`, and links binary modules in `/usr/local/bin`. Most users need the `sudo` option because they lack read-write permissions in `/usr/local`.

- b. **For local installation:** Change directories to your preferred installation folder and use the npm command without `sudo` or `-g`:

```
npm install forceios
```

This command installs Salesforce Mobile SDK in a `node_modules` folder under your current folder. It links binary modules in `./node_modules/.bin/`. In this scenario, you rarely use `sudo` because you typically install in a local folder where you already have read-write permissions.

Android Installation

For the fastest, easiest route to Android development, use the `forcedroid` npm package to install Salesforce Mobile SDK.

1. Use the `forcedroid` package to install the Mobile SDK either globally (recommended) or locally.
 - a. **For global installation:** Append the "global" option, `-g`, to the end of the command. For non-Windows environments, use the `sudo` command:

```
sudo npm install forcedroid -g
```

On Windows:

```
npm install forcedroid -g
```

With the `-g` option, you run `npm install` from any directory. In non-Windows environments, the npm utility installs the package under `/usr/local/lib/node_modules`, and links binary modules in `/usr/local/bin`. Most users need the `sudo` option because they lack read-write permissions in `/usr/local`. In Windows environments, global packages are installed in `%APPDATA%\npm\node_modules`, and binaries are linked in `%APPDATA%\npm`.

- b. For local installation:** Change directories to your preferred installation folder and use the npm command without `sudo` or the `-g` option:

```
npm install forcedroid
```

This command installs Salesforce Mobile SDK in a `node_modules` directory under your current directory. It links binary modules in `./node_modules/.bin/`. In this scenario, you rarely use `sudo` because you typically install in a local folder where you already have read-write permissions.

Uninstalling Mobile SDK npm Packages

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

Uninstalling the Forcedroid Package

The instructions for uninstalling the forcedroid package vary with whether you installed the package globally or locally.

If you installed the package globally, you can run the `uninstall` command from any folder. Be sure to use the `-g` option. On a Unix-based platform such as Mac OS X, use `sudo` as well.

```
$ pwd
/Users/joeuser
$ sudo npm uninstall forcedroid -g
$
```

If you installed the package locally, run the `uninstall` command from the folder where you installed the package. For example:

```
cd <my_projects/my_sdk_folder>
npm uninstall forcedroid
```

If you try to uninstall a local installation from the wrong directory, you'll get an error message similar to this:

```
npm WARN uninstall not installed in /Users/joeuser/node_modules:
"my_projects/my_sdk_folder/node_modules/forcedroid"
```

Uninstalling the Forceios Package

Instructions for uninstalling the forceios package vary with whether you installed the package globally or locally. If you installed the package globally, you can run the `uninstall` command from any folder. Be sure to use `sudo` and the `-g` option.

```
$ pwd
/Users/joeuser
$ sudo npm uninstall forceios -g
$
```

To uninstall a package that you installed locally, run the `uninstall` command from the folder where you installed the package. For example:

```
$ pwd
/Users/joeuser
cd <my_projects/my_sdk_folder>
npm uninstall forceios
```


If you try to uninstall a local installation from the wrong directory, you'll get an error message similar to this:

```
npm WARN uninstall not installed in /Users/joeuser/node_modules:  
"my_projects/my_sdk_folder/node_modules/forceios"
```

Mobile SDK GitHub Repository

More adventurous developers can delve into the SDK, keep up with the latest changes, and possibly contribute to SDK development by cloning the open source repository from GitHub. Using GitHub allows you to monitor source code in public pre-release development branches. In this scenario, both iOS and Android apps include the SDK source code, which is built along with your app.

You don't need to sign up for GitHub to access the Mobile SDK, but we think it's a good idea to be part of 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>

iOS: Cloning the Mobile SDK GitHub Repository (Optional)

1. Clone the Mobile SDK iOS repository to your local file system by issuing the following command at the OS X Terminal app: `git clone git://github.com/forcedotcom/SalesforceMobileSDK-iOS.git`



Note: If you have the GitHub app for Mac OS X, click **Clone in Mac**. In your browser, navigate to the Mobile SDK iOS GitHub repository: <https://github.com/forcedotcom/SalesforceMobileSDK-iOS>.

2. In the OS X Terminal app, change to the directory where you installed the cloned repository. By default, this is the `SalesforceMobileSDK-iOS` directory.
3. Run the install script from the command line: `./install.sh`

Android: Cloning the Mobile SDK GitHub Repository (Optional)

1. In your browser, navigate to the Mobile SDK 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 command prompt in the directory where you installed the cloned repository, and run the install script from the command line: `./install.sh`



Note: Windows users: Run `cscript install.vbs`.

Mobile SDK Sample Apps

Salesforce Mobile SDK includes a wealth of sample applications that demonstrate its major features. Use the hybrid and native samples for iOS and Android 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. You can access them there directly, or you can use an npm command to install them.

Accessing the Samples from GitHub

If you clone Mobile SDK directly from GitHub, all sample files are placed in the correct locations. You can then build the Android samples by including the SalesforceSDK project, SmartStore project, and the sample projects in your Eclipse workspace. For iOS, remember to run `./install.sh` in the repository root folder after cloning the repository. Run the iOS sample projects by opening `SalesforceMobileSDK-iOS/SalesforceMobileSDK.xcworkspace`.

Installing Sample Apps for Android

If you installed the SDK using npm, use the forcedroid command line utility to install the sample apps. You can either:

- Configure your target directory interactively as prompted by the forcedroid app, or
- Specify your target directory directly at the command line.

Specifying the Target Directory Interactively

To enter the target directory interactively, do one of the following:

- If you installed Mobile SDK globally, type `forcedroid samples`.
- If you installed Mobile SDK locally, type `<forcedroid_path>/node_modules/.bin/ forcedroid samples`.

The forcedroid utility prompts you for the target directory name.

```
$node_modules/.bin/forcedroid samples
Enter the target directory of samples: MobileSDKSamples
Adjusting SalesforceSDK library project reference in project.properties.
Renaming application class to FileExplorerApp in source.
Renaming application to FileExplorer in source.
Renaming package name to com.salesforce.samples.fileexplorer in source.
Moving source files to proper package path.
Renaming the app class filename to FileExplorerApp.java.

Your application project is ready in MobileSDKSamples.

To build the new application, do the following:
- cd MobileSDKSamples/FileExplorer
- $ANDROID_SDK_DIR/android update project -p .
- ant clean debug

To run the application, start an emulator or plug in your device and run:
- ant install

To use your new application in Eclipse, do the following:
- Import the forcedroid/native/SalesforceSDK library project,
  and the FileExplorer project into your workspace
- Choose 'Build All' from the Project menu
- Run your application by choosing "Run as Android application"

Before you ship, make sure to plug your OAuth Client ID,
Callback URI, and OAuth Scopes into FileExplorer/res/values/bootconfig.xml.
INFO: SDK directory 'MobileSDKSamples/forcedroid' already exists. Skipping copy.
Adjusting SalesforceSDK library project reference in project.properties
```



Note: For best results, specify a target directory that doesn't already exist. If the target directory doesn't exist, forcedroid creates it. If it exists but doesn't contain the sample directories, forcedroid installs the samples in it. If it exists and already contains one or more of the sample directories, forcedroid exits upon finding an existing directory and doesn't install the rest of the samples.

Android Sample Apps

Native

RestExplorer demonstrates the OAuth and REST API functions of the SalesforceSDK. It's also useful for investigating REST API actions from a Honeycomb tablet.

1. To run the application from your Eclipse workspace, right-click the **RestExplorer** project and choose **Run As > Android Application**.
2. To run the tests, right-click the **RestExplorerTest** project and choose **Run As > Android JUnit Test**.

NativeSqlAggregator demonstrates SQL aggregation with SmartSQL. As such, it also demonstrates a native implementation of SmartStore. To run the application from your Eclipse workspace, right-click the **NativeSqlAggregator** project and choose **Run As > Android Application**.

FileExplorer demonstrates the Files API as well as the underlying Google Volley networking enhancements. To run the application from your Eclipse workspace, right-click the **FileExplorer** project and choose **Run As > Android Application**.

Hybrid

- **AccountEditor**: Demonstrates how to use the SmartSync Data Framework to access Salesforce data.
- **SampleApps/HybridFileExplorer**: Demonstrates the Files API.
- **SampleApps/ContactExplorer**: The **ContactExplorer** sample app uses PhoneGap (also known as Cordova) to retrieve local device contacts. It also uses the `forcetk.mobilesdk.js` toolkit to implement REST transactions with the Salesforce REST API. The app uses the OAuth2 support in Salesforce SDK to obtain OAuth credentials, then propagates those credentials to `forcetk.mobilesdk.js` by sending a javascript event.
- **SampleApps/test/ContactExplorerTest**: Tests for the **ContactExplorer** sample app.
- **SampleApps/VFConnector**: The **VFConnector** sample app demonstrates how to wrap 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, then uses those credentials to set appropriate webview cookies for accessing Visualforce pages.
- **SampleApps/test/VFConnectorTest**: Test for the **VFConnector** sample app.
- **SampleApps/SmartStoreExplorer**: Lets you explore SmartStore APIs.
- **SampleApps/test/SmartStoreExplorerTest**: Tests for the **SmartStoreExplorer** sample app.

Installing Sample Apps for iOS

If you installed the SDK using npm, use the `forceios` command line utility to install the sample apps. You can either:

- Configure your target directory interactively as prompted by the `forceios` app, or
- Specify your target directory directly at the command line.

Specifying the Target Directory Interactively

To enter the target directory interactively, do one of the following:

- If you installed Mobile SDK globally, type `forceios samples`.
- If you installed Mobile SDK locally, type `<forceios_path>/node_modules/.bin/ forceios samples`.

The `forceios` utility prompts you for the target directory name.

```

$forceios samples
Enter the output directory for the samples: mobile_sdk_samples
Staging app dependencies...
Copying SalesforceSDKResources.bundle to /Users/rwhitley/mobile_sdk_samples/RestAPIExplorer/
Copying SalesforceCommonUtils to /Users/rwhitley/mobile_sdk_samples/RestAPIExplorer/
Uncompressing SalesforceOAuth-Release.zip to /Users/rwhitley/mobile_sdk_samples/RestAPIExplorer/
Uncompressing SalesforceSDKCore-Release.zip to /Users/rwhitley/mobile_sdk_samples/RestAPIExplorer/
Copying openssl to /Users/rwhitley/mobile_sdk_samples/RestAPIExplorer/RestAPIExplorer/
Copying sqlcipher to /Users/rwhitley/mobile_sdk_samples/RestAPIExplorer/RestAPIExplorer/
Uncompressing MKNetworkKit-iOS-Release.zip to /Users/rwhitley/mobile_sdk_samples/RestAPIExplorer/RestAPIExplorer/
Uncompressing SalesforceNetworkSDK-Release.zip to /Users/rwhitley/mobile_sdk_samples/RestAPIExplorer/RestAPIExplorer/
Uncompressing SalesforceNativeSDK-Release.zip to /Users/rwhitley/mobile_sdk_samples/RestAPIExplorer/RestAPIExplorer/
Dependencies copied successfully!
Staging app dependencies...
Copying SalesforceSDKResources.bundle to /Users/rwhitley/mobile_sdk_samples/NativeSqlAggregator/
Copying SalesforceCommonUtils to /Users/rwhitley/mobile_sdk_samples/NativeSqlAggregator/
Uncompressing SalesforceOAuth-Release.zip to /Users/rwhitley/mobile_sdk_samples/NativeSqlAggregator/
Uncompressing SalesforceSDKCore-Release.zip to /Users/rwhitley/mobile_sdk_samples/NativeSqlAggregator/
Copying openssl to /Users/rwhitley/mobile_sdk_samples/NativeSqlAggregator/NativeSqlAggregator/
Copying sqlcipher to /Users/rwhitley/mobile_sdk_samples/NativeSqlAggregator/NativeSqlAggregator/
Uncompressing MKNetworkKit-iOS-Release.zip to /Users/rwhitley/mobile_sdk_samples/NativeSqlAggregator/NativeSqlAggregator/
Uncompressing SalesforceNetworkSDK-Release.zip to /Users/rwhitley/mobile_sdk_samples/NativeSqlAggregator/NativeSqlAggregator/

```



Note: For best results, specify a target directory that doesn't already exist. If the target directory doesn't exist, forceios creates it. If it exists but doesn't contain the sample directories, forceios installs the samples in it. If it exists and already contains one or more of the sample directories, forceios exits upon finding an existing sample directory and doesn't install the samples.

iOS Sample Apps

Native

- **RestAPIExplorer** exercises all of the native REST API wrappers. It resides in the Mobile SDK for iOS under `native/SampleApps/RestAPIExplorer`.
- **NativeSqlAggregator** shows SQL aggregation examples as well as a native SmartStore implementation. It resides in the Mobile SDK for iOS under `native/SampleApps/NativeSqlAggregator`.
- **FileExplorer** demonstrates the Files API as well as the underlying MKNetwork network enhancements. It resides in the Mobile SDK for iOS under `native/SampleApps/FileExplorer`.

Hybrid

- **AccountEditor**: Demonstrates how to use the SmartSync Data Framework to access Salesforce data.
- **HybridFileExplorer**: Demonstrates the Files API.
- **ContactExplorer**: The ContactExplorer sample app uses PhoneGap (also known as Cordova) to retrieve local device contacts. It also uses the `forcetk.mobilesdk.js` toolkit to implement REST transactions with the Salesforce REST API. The app uses the OAuth2 support in Salesforce SDK to obtain OAuth credentials, then propagates those credentials to `forcetk.mobilesdk.js` by sending a JavaScript event.
- **VFConnector**: The VFConnector sample app demonstrates how to wrap 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, then uses those credentials to set appropriate webview cookies for accessing Visualforce pages.
- **SmartStoreExplorer**: Lets you explore SmartStore APIs.

What's New

Salesforce developer tools have made significant strides since Dreamforce '12. Changes to Salesforce Mobile SDK as well as Visualforce make Salesforce mobile development easier and accessible to all levels of developers.

What's New in Mobile SDK 2.0

Top new features in Mobile SDK 2.0 include:

- Highly flexible mobile architecture
 - ◇ OAuth2 Authentication can now be handled on-demand, enabling mobile apps to start un-authenticated.
 - ◇ Mobile apps can now authenticate against standard force.com and community licenses.
- SmartSync data framework allow developers to work with real objects by encapsulating REST APIs.
 - ◇ Write substantially less code for CRUD operations.
 - ◇ SmartSync can also seamlessly integrate into the SmartStore offline storage database for storing data offline without additional code
- Node Packaged Module (NPM) installers.
 - ◇ Salesforce provides two packages: forceios for the iOS Mobile SDK, and forcedroid for the Android Mobile SDK. These packages provide a static snapshot of an SDK release.

Create new native and hybrid apps directly from the command line with a simple script.

General SDK Improvements

- Refresh tokens are now explicitly revoked from the server upon logout.
- Added support for community users to login.
- Consolidated our Cordova JS plugins and utility code into one file (`cordova.force.js`).
- Updated `forcetk.js` and renamed to `forcetk.mobilesdk.js`, to pull in the latest functionality from ForceTK and enhance its ability to work with the Mobile SDK authentication process.
- Fixed session state rehydration for Visualforce apps, in the event of session timeouts during JavaScript Remoting calls in Visualforce.
- SmartStore now supports 'Smart SQL' queries, such as complex aggregate functions, JOINS, and any other SQL-type queries.
- NativeSqlAggregator is a new sample app to demonstrate usage of SmartStore within a native app to run Smart SQL queries, such as aggregate queries.
- SmartStore now supports three data types for index fields – 'string', 'integer', and 'floating'.

iOS

- All projects and template apps have been converted to use ARC.
- All Xcode projects are now managed under a single workspace (`SalesforceMobileSDK.xcworkspace`). This allows for a few benefits:
 - ◇ During development of the SDK, any changes to underlying `libraries/projects` will automatically be reflected in their consuming projects/applications.
 - ◇ You can now debug all the way through the stack of dependencies, from a sample app down through authentication and other core functionality.

- ◊ No more “staging” of binary artifacts as a prerequisite to working with the projects. `install.sh` now simply syncs the sub-modules of the repository, after which you’re free to start working in the workspace.
- Native and mobile template apps no longer rely on parent app delegate classes to successfully leverage OAuth authentication. This means that you the developer are free to form up your `AppDelegate` app flow however you choose, leveraging the updated `SFAuthenticationManager` component wherever it’s practical to do so in your app.
- All hybrid dependencies are now decoupled from `SalesforceHybridSDK.framework` (which has been retired). This means you can now mix and match your own versions of Cordova, openssl, etc., in your app. The core functionality of the framework itself has now been converted into a static library.

Android

- Extending `ForceApp` or `ForceAppWithSmartStore` is no longer a requirement.
- Added `SalesforceListActivity` and `SalesforceExpandableListActivity`, that provides Salesforce wrappers around the standard `ListActivity` and `ExpandableListActivity` classes, respectively.
- `rest.xml` has been removed and replaced with `bootconfig.xml`. Login options do not have to be supplied through code anymore, they can be specified in `bootconfig.xml`.
- Bootstrap flow for hybrid apps now occurs on the native side, thereby enabling faster initial load times (`bootstrap.html` is no longer required or used).
- Changed the package name of the SmartStore library, so that it can coexist with other library projects.

What's New in Mobile SDK 2.1

Salesforce Mobile SDK 2.1 builds on the substantial advances of 2.0. As of this printing, the full list of new features was not finalized, but here are some highlights:

- Files API—Wraps the file requests in Salesforce REST API. Available in iOS native, Android native, and hybrid SDKs.
- Push notifications—Limited support for the push notification Developer Preview feature in the Winter ’14 release.
- Flexible SmartSync endpoints—SmartSync now supports custom REST API endpoints as well as Apex REST objects.
- Networking enhancements—As a result of the Files API development, networking libraries have changed. In iOS, Mobile SDK now uses the `MKNetworkSDK` for REST request network operations. Android now uses the Google Volley library. These libraries give apps a performance boost and developers extra control over the request queue.
- npm Samples Installer—You can now use the `forceios` and `forcedroid` apps to install the sample applications.

You can find the official What’s New for Mobile SDK 2.1 at http://wiki.developerforce.com/page/Mobile_SDK_Release_Notes.

HTML5 Improvements in Visualforce (Winter ’14 Release)

Visualforce features a number of improvements in Winter ’14 that are intended to make HTML5 development easier. See the Winter ’14 Release Notes for full details.

New `<apex:input>` Component

`<apex:input>` is a new, HTML5-friendly, general purpose input component that adapts to the data expected by a form field. It uses the `HTML type` attribute, particularly values new in HTML5, to allow client browsers to display type-appropriate user input widgets, such as a date picker or range slider, or use a type-specific keyboard on touch devices, or to perform client-side formatting or validation, such as with a numeric range or a telephone number.

type Attribute for <apex:input> and <apex:inputField>

Use the new `type` attribute with `<apex:input>` and `<apex:inputField>` to display input user interface elements using HTML5 browser features or JavaScript user interface widgets, to supplement or replace the default Salesforce input elements.

HTML5 <datalist> Element for Input Components

The HTML5 `<datalist>` element specifies a list of values to associate with an `<input>` element, which can be used to show a list of completion options filtered by input. Use the new `list` attribute for the `<apex:input>`, `<apex:inputField>`, and `<apex:inputText>` components to generate a `<datalist>` block for the associated input field.

Additional Components with Support for Pass-Through HTML Attributes

You can add arbitrary attributes to many Visualforce components that will be “passed through” to the rendered HTML. This is useful, for example, when using Visualforce with JavaScript frameworks, such as jQuery Mobile, AngularJS, and Knockout, which use `data-*` or other attributes as hooks to activate framework functions. It can also be used to improve usability with HTML5 features such as placeholder “ghost” text, pattern client-side validation, and title help text attributes.

Accessibility Improvements to Visualforce Components

The accessibility of Visualforce pages has been improved, by improving the semantics and completeness of the HTML generated by a number of Visualforce components.

Deferred Loading of JavaScript Resources via <apex:includeScript>

The `<apex:includeScript>` component has a new attribute, `loadOnReady`, which controls whether a JavaScript resource loaded by the component is loaded immediately (existing behavior), or is delayed until the document model (DOM) for the page is constructed.

Chapter 3

HTML5 and Hybrid Development

In this chapter ...

- [Getting Started](#)
- [HTML5 Development Tools](#)
- [Delivering HTML5 Content With Visualforce](#)
- [Accessing Salesforce Data: Controllers vs. APIs](#)
- [Introduction to Hybrid Development](#)
- [Hybrid Apps Quick Start](#)
- [Guidelines and Tips for Hybrid Apps](#)

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 Force.com platform by:

- Using Visualforce to deliver the HTML content
- Using JavaScript remoting to invoke Apex controllers for fetching records from Force.com

In addition, you can repurpose HTML5 code in a standalone Mobile SDK hybrid app, and then distribute it through an app store. Converting to hybrid involves creating a Mobile SDK project to get the native container, then importing your HTML5 files into the 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 native mobile container and distribute it in the Apple® AppStore® 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 currently available industry libraries that you can leverage 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 Force.com organization.
- Some knowledge of Apex and Visualforce is necessary.



Note: This type of development uses Visualforce. You can't use Database.com.

HTML5 Development Tools

Salesforce provides a suite of tools that makes HTML5 development surprisingly simple. Some of these tools are built on popular open source JavaScript frameworks, while others are home-grown solutions. Also, a group of third-party, open source Mobile Packs brings the power of industry-standard architectures to Salesforce app development.

Mobile Design Templates

Mobile design templates are an open-source library of 22 visually striking, mobile-optimized HTML5 and CSS3 markup for common mobile use cases for interacting with Salesforce data.

- The templates are modular, customizable, open-source CSS3 and HTML5 markup that can be modified at will to meet the specific UI/UX requirements of a mobile app.
- You can combine these static templates with a JavaScript library like ForceTk or one of the Mobile Packs for Backbone, Angular or Knockout to provide live data bindings with any Salesforce backend.
- The templates provide cross-platform (iOS, Android etc.) support, courtesy of the use of standard Web technologies like HTML5, CSS3 and JavaScript.

- All templates are optimized for the phone form factor.
- The base HTML5/CSS3 can be modified to work with any Salesforce object (standard or custom) in the context of any mobile use case.

There are templates to view and edit customer data, view backend reports, find nearby records, and much more. These starting points for UI and UX design should dramatically shorten the time it takes to develop a great looking Web or hybrid app on the Salesforce Platform. Thereafter, you can customize and reuse these templates at will to meet the specific requirements of your mobile app.

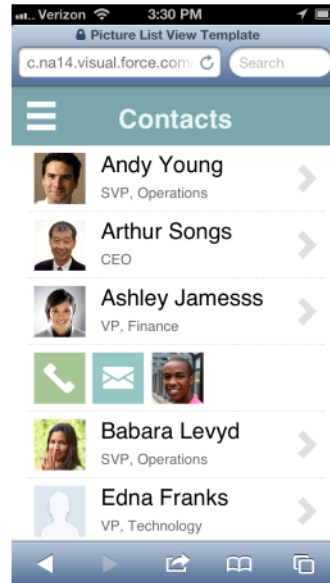
Mobile templates cover the following popular use cases:

- [List View Templates List View](#)—Provides different visual representations for showing a list of standard or custom Salesforce records.
- [Detail View Templates Detail View](#)—Provides various read-only views of a standard or custom data record. Typically, users will navigate to one of these detail views by clicking a record in a List View.
- [Data Input Templates Data Input](#)—Used to capture user input from a phone. The different form elements included in these templates (phone, date, number, text, etc.) can be used in any mobile app that requires users to add or update Salesforce data
- [Map View Templates Map View](#)—Provides Google Map-based designs for implementing the common ‘Find Nearby’ functionality on a mobile device. These templates can be combined with the Geolocation custom field in Salesforce, and its corresponding SOQL companion, to add geolocation functionality to any Web or hybrid mobile app.
- [Calendar View Templates Calendar](#)—Provides mobile optimized views of a user’s Salesforce calendar (Tasks and Events).
- [Report and Dashboard Templates Report and Dashboard](#)—Provides mobile optimized report views of Salesforce data. These templates are developed using the open-Source D3 charting library and developers can combine them with the Salesforce Analytics API to add reporting capabilities to their mobile apps.
- [Miscellaneous Templates Miscellaneous](#)—Use these templates as a starting point to add a Settings, Splash or About screen to your mobile app.

HTML5 Mobile Templates Sample App

The mobile templates are described in the context of a simple mobile Web app created in Visualforce using the Picture List View template to view contact records in Salesforce. List View templates provide different visual representations for showing a list of standard or custom Salesforce records and the Picture List View template in particular can be used to display any data that has a picture associated with it (contact, user, product, etc.).

The template also has an optional feature whereby a swipe-right on any image in the list reveals some Quick Action icons. The user can then perform these quick actions (like emailing or calling the contact) directly from this view (versus from a drill-down detail view).



You can also peruse the entire codebase for this sample app in the Sample Apps directory of the GitHub repo. Throughout this section, you can follow along as this app is built into a fully featured mobile app.



Note: While the Picture List View template and this sample app use contacts as an example, every mobile design template is use-case agnostic and can be modified and reused with any Salesforce object (standard or custom) in the context of any mobile use case.

Using Mobile Design Templates in Visualforce

1. Download the [templates project from GitHub](https://github.com/developerforce/Mobile-Design-Templates.git). You can do so by executing the following from the command line.

```
git clone https://github.com/developerforce/Mobile-Design-Templates.git
```



Note: If you don't have Git installed, you can also click the **Download ZIP** icon on the right of the repo home page and download the project as a zip file.

2. Upload the templates zip file as a Static Resource in your DE Org.
3. Import the necessary JavaScript and CSS from the templates Static Resource zip file using the following Visualforce page skeleton.

```
<apex:page docType="html-5.0"
    showHeader="false"
    sidebar="false"
    standardStylesheets="false"
    standardController="Contact"
    extensions="Contacts_Ext">
  <head>
    <meta charset="utf-8"/>
    <meta name="viewport" content="width=device-width,
      initial-scale=1, minimum-scale=1, maximum-scale=1,
      user-scalable=no"/>
    <apex:stylesheet value=
      "{!URLFOR($Resource.Mobile_Design_Templates,
        'Mobile-Design-Templates-master/common/css/app.min.css')}" />
    <apex:includeScript value=
      "{!URLFOR($Resource.Mobile_Design_Templates,
```

```

        'Mobile-Design-Templates-master/common/js/jquery2.0.2.min.js') }"/>
    <apex:includeScript value=
        "{!URLFOR($Resource.Mobile_Design_Templates,
        'Mobile-Design-Templates-master/common/js/jquery.touchwipe.min.js') }"/>
    <apex:includeScript value=
        "{!URLFOR($Resource.Mobile_Design_Templates,
        'Mobile-Design-Templates-master/common/js/main.min.js') }"/>
</head>
<body>
    <div id="mainContainer"/>
</body>
</apex:page>

```

The two most important imports for using the mobile templates in a Web page are `app.min.css` and `main.min.js` (note that the GitHub project also has the uncompressed versions of these files for easier review). These two files, respectively, define the CSS and minimal JavaScript required to render the templates. In addition to these two files, the templates also require jQuery 2.0.2 (for basic DOM manipulation in `main.min.js`). The Picture List View Template used in this sample also requires a small JavaScript library (`jquery.touchwipe.min.js`) to enable touch gestures in the List View. In addition to the JavaScript and CSS imports, note the use of the HTML5 doctype and the disabling of the standard stylesheets, header, and sidebar in the Visualforce page. This is a way to optimize the page to load faster on a mobile device.

Data Binding with Mobile Templates

All Mobile Design Templates in the GitHub repo use static, hard-coded data for illustrative purposes. This helps a new developer quickly review the overall UI/UX offered by a template and determine if it's a good starting point for their own mobile use case or app. However, in order to use a template in an actual mobile app, the HTML5/CSS3 markup has to be populated with live data from Salesforce. In the case of this sample app, we need to query a list of contact records from Salesforce and then build out the appropriate page markup. There are several options for a developer to bind these mobile Web templates to Salesforce data.

- JavaScript Remoting, when the templates are used in a Visualforce page.
- JavaScript wrapper for the REST API (or ForceTK) to perform CRUD access to Salesforce data from a template Visualforce or Web page.
- Mobile Packs for Backbone, Angular or Knockout, when the templates are used in an app built using one of those MV* frameworks.

In the interest of keeping things simple, we've used JavaScript Remoting in this sample to query the list of contact records. Before we dive deeper into the data binding code, let's quickly review how we store the contact pictures in Salesforce. Since the contact standard object does not have a native picture field, we created a rich text custom field (**Contact_Pic__c**) and used it to upload thumbnail images to contact records. You can also upload images as attachments or Chatter files, then use a combination of formula fields and trigger or API logic to store and display the images.

Next, review how to query and display a dynamic list of contact records using the Picture List View Template.

Using JavaScript Remoting to Query Contact Records

The following markup shows how to use JavaScript Remoting to provide the data binding for the page.

```

var contactRecs = new Array();
var compiledListViewTempl = _.template($("#listView").html());

$(document).ready(function() {
    getAllContacts();
});

```

```

function getAllContacts(){
    Visualforce.remoting.Manager.invokeAction(
        '{!$RemoteAction.Contacts_Ext.getContactRecs}',
        function(records, e) {
            showContacts(records);},
        {escape:false});
}

function showContacts(records) {
    contactRecs.length = 0;
    for(var i = 0; i < records.length; i++) {
        records[i].Pic = '{!URLFOR($Resource.BlankAvatar)}';
        if (typeof records[i].Contact_Pic__c != "undefined"){
            records[i].Pic = $(records[i].Contact_Pic__c).attr('src');
        }
        contactRecs[records[i].Id] = records[i];
    }

    $('#mainContainer').empty();
    $('#mainContainer').append(compiledListViewTempl({contacts : records}));
    $(document).trigger('onTemplateReady');
}

```

The `getAllContacts` JavaScript method invokes the `getContactRecs` function on the `Contacts_Ext` Extension class via JavaScript Remoting. The method returns a list of contact records that are then processed in the `showContacts` callback method. There, we iterate the contact records and assign a default 'blank avatar' image to any contact record that doesn't have an associated image in the `Contact_Pic__c` rich text custom field (lines 18-24). Finally, we insert the list of contacts into the page DOM (lines 2, 27) using the Underscore utility library (more on this later). Note the triggering of the `onTemplateReady` custom JavaScript event on line 28. As mentioned earlier, the templates use a minimal amount of JavaScript (`main.min.js`) to enable basic user interactivity. The `main.min.js` script listens for the `onTemplateReady` event before executing some initialization logic and you're therefore required to fire that event once the template markup has been inserted into the page DOM.

Using Underscore to Generate the Template Markup

The real action in the Visualforce page happens in the Underscore template that generates the dynamic list of contacts using markup from the Picture List View Template.

```

<script type="text/html" id='listView'>
  <div class="app-wrapper">

    <nav class="main-menu">
      <a href="#">Accounts</a>
      <a href="#">Opportunities</a>
    </nav>

    <header>
      <div class="main-menu-button main-menu-button-left"><a class="menu">&nbsp;</a></div>

      <h1>Contacts</h1>
    </header>

    <div class="app-content">
      <ul id="cList" class="list-view with-swipe left-thumbs right-one-icons">
        <% for(var i = 0; i < contacts.length; i++){ %>
          <li>
            <div class="thumbs">
              <% if (typeof(contacts[i].Phone) != "undefined") { %>
                <a href="tel:<%= contacts[i].Phone %>" class="thumb thumb-1">
                  <img class="thumb" src=
                    '{!URLFOR($Resource.Mobile_Design_Templates,
                      Mobile-Design-Templates-master/common/images/icons/tile-phone.png}'

```

```

    ) }"/>
  </a>
<% } %>

<% if (typeof(contacts[i].Email) != "undefined") {%>
  <a href="mailto:<%= contacts[i].Email %>" class="thumb thumb-2">
    <img class="thumb" src=
      "{!URLFOR($Resource.Mobile_Design_Templates,
        'Mobile-Design-Templates-master/common/images/icons/tile-email.png'
      ) }"/>
  </a>
<% } %>
  
</div>
<a href="#/contact/<%= contacts[i].Id %>" class="content">
  <h2><%= contacts[i].Name %></h2>
  <%= contacts[i].Title %>
  <div class="list-view-icons">
    <span class="icon-right-arrow">&nbsp;</span>
  </div>
</a>
</li>
<% } %>
</ul>
</div>
</div>
</script>

```

HTML5 and CSS3 markup is used in this template. This is our first glimpse of the Mobile Design Templates in action. Just copy and paste this markup from the Picture List View Template in GitHub (minus the dynamic binding part), and make a couple of minor tweaks to the markup to suit your specific use case. For example, removing the gear icon from the header and updating the menu list (under the `<nav class="main-menu">` section).

As is evident from the above markup, all templates use basic HTML5 with CSS3 styling to generate a mobile-optimized view. They have no dependency on external frameworks like jQuery Mobile, Twitter Bootstrap, etc. The template markup and CSS is also very modular and composable. Adding a list item in the Picture List View Template is as simple as adding a `` tag with the list-view, swipe-left thumbs, and right-one-icons CSS styles applied to it (line 15). Adding a Quick Action icon (revealed when the user does a swipe-right on the thumbnail) is simply a matter of applying the thumb CSS style to the `` tag (lines 21, 28). And so on. For a more detailed breakdown of the different components of the Picture List View and other templates, simply scroll down through the interactive home page and see the templates come to life.

Remember also that the templates are not an all-or-nothing monolith. You can and should pick and choose components of a particular template that fit your specific use case and requirements. Don't want the contact title to show in your list view? Just drop the respective markup from the page. Don't need the Quick Action on-swipe component? Simply drop the entire `<div class="thumbs">` tag.

Customizing Look and Feel

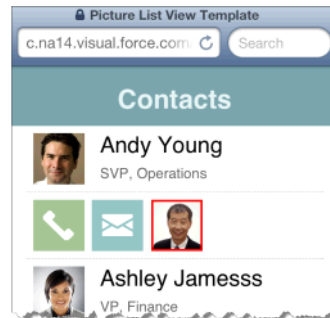
In addition to being modular and composable, the look and feel of the templates are also completely customizable. Simply modify [app.css](#) to implement your unique styling and UX requirements. Let's take a look at a very simple example to see this in action. For example, we wanted to display a red border around the contact image when the user does a swipe-right to reveal the Quick Action icons. All we have to do is to change the respective style class in `app.css`.

```

ul.list-view li.swiped .thumb-3 {
  left: 115px;
  border:2px solid red;
}

```

And voila, the list view looks a little different. The CSS used in the templates was generated from the cloud, so if you're more comfortable in a tool like Compass, you can modify the cloud files instead.



List View Templates

List view templates provide different visual representations for showing a list of standard or custom Salesforce records. The following views are supported:

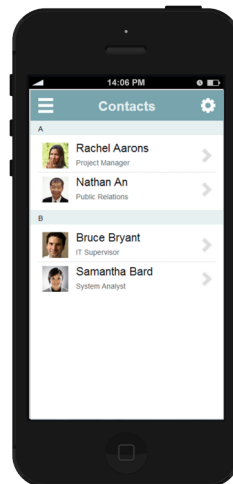
- [Picture](#)
- [Collapsible](#)
- [Standard](#)
- [Tabbed](#)
- [Carousel](#)
- [Timeline](#)

Picture

The Picture List View template can be used to display any data that has a picture associated with it (contact, user, product, etc.). The MyContacts Visualforce page of the Mobile Templates Jumpstarter sample app shows an example of this template in action.



Note: The template shows a list of contact records as an example. Since the contact standard object does not have a native picture field, this example assumes that pictures are stored as attachment records or Content/Chatter Files.



This template is composed of a Header and List Elements component.

Header

This is a common, shared component used in all of the Mobile Design Templates.

```
<nav id="main-menu" class="main-menu">
  <a href="#">Something</a>
  <a href="#">Something</a>
  <a href="#">Something</a>
</nav>

<header>
  <div id="main-menu-button-left"
    class="main-menu-button main-menu-button-left">
    <a class="menu">&nbsp;</a>
  </div>
  <div id="main-menu-button-right"
    class="main-menu-button main-menu-button-right">
    <a class="gear">&nbsp;</a>
  </div>
  <h1>Contacts</h1>
</header>
```

The `<nav>` element should contain all the items to be displayed in the sliding Menu panel (click the menu icon on the right to see it in action). Icons can be displayed left (`main-menu-button-left`) or right (`main-menu-button-right`) aligned by applying the respective CSS class to the `<div>` tag. The templates have a small set of basic icons defined for this header component (Menu, Gear, Left Arrow, Right Arrow). However, developers are free to create their own custom icons and add them to the header component.

List Elements

Each list element in this template can be added with the following markup.

```
<ul class="list-view left-thumbs right-one-icons">
  <li>
    <div class="thumbs">
      
    </div>
    <a href="#" class="content">
      <h2>Rachel Aarons</h2>
      Project Manager
      <div class="list-view-icons">
        <span class="icon-right-arrow">&nbsp;</span>
      </div>
    </a>
  </li>
</ul>
```

Adding Quick Action Icons with a Swipe Gesture

An optional feature of this Picture List View template is the ability to display some Quick Action icons when the user swipes any picture in the list. The user can then perform these quick actions (like emailing or calling the contact) directly from this view (versus from a drill-down detail view).

This optional functionality can be enabled by adding the `with-swipe` CSS style to the `` element, and adding the appropriate quick action links and icons to the `<div class="thumbs">` section.

```
<ul class="list-view left-thumbs right-one-icons">
  <li>
    <div class="thumbs">
      
    </div>
    <a href="#" class="content">
```



```

        <h2>Rachel Aarons</h2>
        Project Manager
        <div class="list-view-icons">
            <span class="icon-right-arrow">&nbsp;</span>
        </div>
    </a>
</li>
</ul>

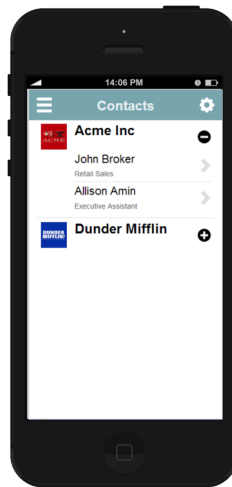
```



Note: This swipe gesture support is enabled by a small amount of JavaScript. Look for the `setupSwipeList` function in `main.js` for more details.

Collapsible

The Collapsible List View template can be used to display grouped and summarized list data. Users can use an accordion-style control to collapse and expand each group of data. The sample template shows an example of contact records grouped by account, but the template can be used for any use case that requires a grouped view of list data.



This template is composed of the following components.

Collapsible Section

Use the following markup to display a collapsible section with a thumbnail image.

```

<ul class="list-view collapsable left-thumbs right-one-icons">
    <li>
        <div class="thumbs">
            
        </div>
        <div class="content">
            <h1>Acme Inc</h1>
            <div class="list-view-icons">
                <span class="icon-plus">&nbsp;</span>
            </div>
        </div>
    </li>
</ul>

```

List Elements

In order to add list items to a collapsible section, add the following markup to the parent `` tag.

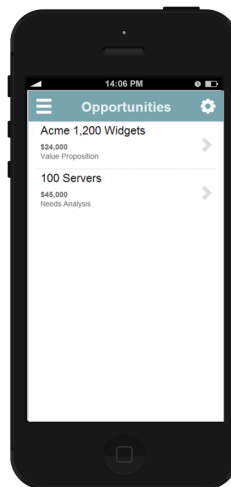
```
<ul class="list-view right-one-icons">
  <li>
    <div class="content">
      <h2>John Broker</h2>
      Retail Sales
      <div class="list-view-icons">
        <span class="icon-right-arrow">&nbsp;</span>
      </div>
    </div>
  </li>
  <li>
    <div class="content">
      <h2>Allison Amin</h2>
      Executive Assistant
      <div class="list-view-icons">
        <span class="icon-right-arrow">&nbsp;</span>
      </div>
    </div>
  </li>
</ul>
```



Note: The accordion control uses a small amount of JavaScript. Look for the `setupCollapsibleMenus` function in `main.js` for more details.

Standard

The Standard List View template shows a simple list of data. Each list element can display one or more lines of data, depending on the specific use case. This template is composed of the following components.



List Elements

Each list element (with the right arrow indicating a clickable item) can be added via the following markup.

```
<ul class="list-view right-one-icons">
  <li>
    <a href="#" class="content">
      <h2>Acme 1,200 Widgets</h2>
      <div class="list-view-icons">
        <span class="icon-right-arrow">&nbsp;</span>
      </div>
    </a>
  </li>
</ul>
```

```

        </a>
      </li>
    <li>
      <a href="#" class="content">
        <h2>100 Servers</h2>
        <div class="list-view-icons">
          <span class="icon-right-arrow">&nbsp;</span>
        </div>
      </a>
    </li>
  </ul>

```

Add additional markup to the `` element to display additional information for each list element. For example, the following markup adds the opportunity amount and stage to the list view.

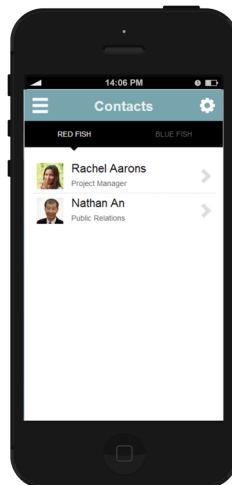
```

<li>
  <a href="#" class="content">
    <h2>Acme 1,200 Widgets</h2>
    <p>
      <strong>$24,000</strong><br/>
      Value Proposition
    </p>
    <div class="list-view-icons">
      <span class="icon-right-arrow">&nbsp;</span>
    </div>
  </a>
</li>

```

Tabbed

The Tabbed List View template shows list data organized under two or more tabs. Users can quickly navigate across the different tabs by clicking them. For example, this template can be used to show a list of open and closed cases in a single list view.



In addition to the common Header component, this template is composed of the following components.

Tab

The following markup adds tabs to the view.

```

<div id="tabbed-list-view-nav" class="tabbed-list-view-nav">
  <a href="#" class="span-50 on">Red Fish</a>
  <a href="#" class="span-50">Blue Fish</a>

```

```
<div id="tabbed-list-view-nav-arrow" class="tabbed-list-view-nav-arrow">&nbsp;</div>
</div>
```

Users can navigate across the tabs by clicking them (try it on the right). This interactivity requires a small amount of JavaScript (see the `setupTabbedLists` function in `main.js` for details).



Note: The appropriate CSS style element has to be applied to each `<a>` tag, depending on the number of tabs to be displayed (up to a max of four tabs). For example, `span-50` should be used when displaying two tabs, `span-33` should be used when displaying three tabs, and `span-25` should be used when displaying four tabs.

List Elements

In order to add list items to a tab, create an unordered list `` element (with the `tabbed-list-view` CSS class), and insert a list-item element (``) for each tab you want to create. You can also nest an unordered list `` within each tab `` element to display a list of data corresponding to each respective tab.

```
<ul id="tabbed-list-view" class="tabbed-list-view">
  <li>
    <ul class="list-view left-thumbs right-one-icons">
      <li>
        <div class="thumbs">
          
        </div><!-- .thumbs -->
        <a href="#" class="content">
          <h2>Rachel Aarons</h2>
          Project Manager
          <div class="list-view-icons">
            <span class="icon-right-arrow">&nbsp;</span>
          </div>
        </a>
      </li>
      <li>
        <div class="thumbs">
          
        </div><!-- .thumbs -->
        <a href="#" class="content">
          <h2>Nathan An</h2>
          Public Relations
          <div class="list-view-icons">
            <span class="icon-right-arrow">&nbsp;</span>
          </div>
        </a>
      </li>
    </ul>
  </li>
</ul>
```

Carousel

The Carousel List View template provides a touch-optimized carousel view of images. Users can cycle through the images by using swipe gestures on their mobile phone. For example, this template can be used to allow users to browse through a set of products stored in Salesforce. The product images can be stored in Salesforce as either attachment records or Content/Chatter files.



This template uses the open-source SwipeView library to implement the carousel control. In addition to the common Header component, this template is composed of the following components.

Carousel

Most of the work in this template is done in JavaScript by the SwipeView library. The page markup simply needs a parent `<div>` tag to host the carousel control.

```
<div id="app-content">
  <div id="carousel-wrapper"></div>
</div>
```

The SwipeView library can then be initialized in JavaScript with the set of images (including width and height) to display in the carousel. Here's a small JavaScript snippet showing how to initialize the carousel:

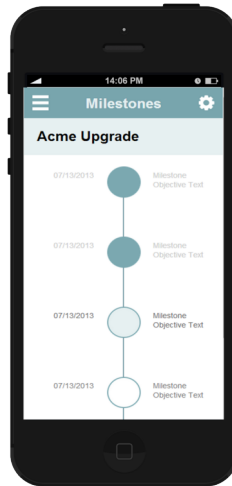
```
<script type="text/javascript">
$(document).ready(function() {
  var slides = [
    {
      img: 'images/placeholders/gallery/pic01.png',
      width: 1017,
      height: 699,
      desc: '<h3>A Product</h3>$341.12'
    }
  ];

  carouselObj.init({
    container: "#carousel-wrapper",
    slides: slides
  });
});
</script>
```

In a real-world app, the images will typically be retrieved from Salesforce using an API call or JavaScript remoting.

Timeline

The Timeline List View template can be used to show a timeline view of data in Salesforce. For example, this template can be used to show a set of Milestones associated with a Project, shipping status for an order or opportunity stage history. The ProjectMilestones Visualforce page in the Mobile Templates Jumpstarter sample app shows an example of this template in action.



All time points are added as `` elements to the following `` element.

```
<ul class="list-view list-view-milestones">
</ul>
```

In addition to the common Header component, this template is composed of the following components.

Completed

Add the complete CSS style class to the `` child element to display a completed/past time-point in the timeline.

```
<li class="complete">
  <div class="content">
    <div class="date">07/13/2013</div>
    <div class="objective">Milestone Objective Text</div>
  </div>
</li>
```

As shown above, the date and objective CSS classes can be used to display the date and text associated with the time-point.

Current

Add the current CSS style class to the `` child element to display the current time-point.

```
<li class="current">
  <div class="content">
    <div class="date">07/13/2013</div>
    <div class="objective">Milestone Objective Text</div>
  </div>
</li>
```

Future

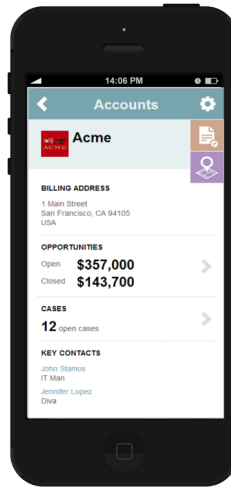
Add the future CSS style class to the `` child element to display future time-points.

```
<li class="future">
  <div class="content">
    <div class="date">07/13/2013</div>
    <div class="objective">Milestone Objective Text</div>
  </div>
</li>
```

Detail View Templates

Detail view templates provide various read-only views of a standard or custom data record. Typically, users will navigate to one of these detail views by clicking a record in a list view. There are three detail view templates provided.

Detail View 1



In addition to the common [Header](#) component, this template is composed of the following components.

Banner Section

This component should be used to display the most important information related to the record. For example, it shows the icon and name associated with an account record. It's assumed the Icon is stored as an attachment related to the account record.

```
<div class="detail-view-header left-thumb">
  <div class="content">
    
    <h1>Acme</h1>
  </div>
</div>
```

Quick Action Menu

This optional component provides a limited set of Quick Actions that a user can perform on a specific record type. The specific set of Quick Actions depend on the sObject and mobile use case and is therefore left to the developer. For example, since this template shows an account record, it includes Quick Action icons for taking a note and showing the account address on the mobile map application (a common mobile use case for account data).

Quick Action links can be added to the banner section with the following markup.

```
<div class="detail-view-header left-thumb with-action-panel">
  <div class="content">
    
    <h1>Acme</h1>
    <div class="detail-view-action-panel">
      <a href="#" class="action-note open-modal">&nbsp;</a>
      <a href="#" class="action-map open-modal">&nbsp;</a>
    </div>
  </div>
</div>
```

By default, clicking any Quick Action icon shows a simple modal dialog box. (Click any of the icons on the right for an example.) Developers should customize the actual action that occurs with each Quick Action, depending on the specific mobile app and use case.

Chatter Publisher Actions are a perfect complement to this concept of Quick Actions. You can define an object-specific or Global Publisher Action in Salesforce (for example, quick create of an opportunity record), and then invoke that [Publisher Action via the Salesforce REST API](#) when a user clicks the corresponding Quick Action on the detail view.

Data Section

This component can be used to display selected fields from its respective sObject. For example, the following markup shows the Billing Address for an account record.

```
<section class="border-bottom">
  <div class="content">
    <h3>Billing Address</h3>
    <p>
      1 Main Street<br/>
      San Francisco, CA 94105<br/>
      USA
    </p>
  </div>
</section>
```

Summary Section

This component can be used to display important roll-up summary data from one or more related child objects. For example, it shows the total number of open cases related to an account. A summary section can be made clickable via an <a> link and developers can navigate users to a list of related child records (using one of the list view templates) on click.

```
<ul class="list-view right-one-icons large-padding">
  <li>
    <a href="#" class="content">
      <h3>Cases</h3>
      <h1 class="inline">12</h1> open cases
      <div class="list-view-icons">
        <span class="icon-right-arrow">&nbsp;</span>
      </div>
    </a>
  </li>
</ul>
```

Related List Section

This component can be used to display related child records. For example, it shows a list of contact records related to an account.

```
<section class="border-top">
  <div class="content">
    <h3>Key Contacts</h3>
    <p>
      <a href="#">John Stamos</a><br/>
      IT Man
    </p>
    <p class="no-bottom-padding">
      <a href="#">Jennifer Lopez</a><br/>
      Diva
    </p>
  </div>
</section>
```


Each related child record can be made clickable via a `<a>` link and the user can be directed to the respective detail view for that child record on click.

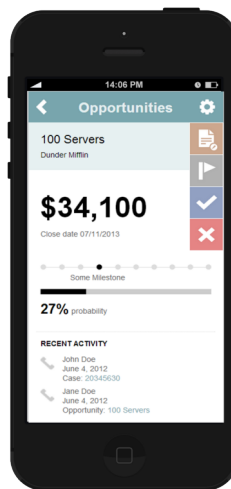
Action Buttons

This component can be used to display one or more action buttons (like Edit, Delete, etc.).

```
<section class="data-capture-buttons one-buttons">
  <div class="content">
    <a href="#">edit</a>
  </div>
</section>
```

Similarly, the `two-buttons` style can be used to display two buttons in this section.

Detail View 2



In addition to the common [Header](#) component, this template is composed of the following components.

Banner Section

Similar to the Banner Section in Detail View 1, this component is used to display the most important information related to the record. For example, this template shows the opportunity name and account associated with an opportunity record.

Quick Action Menu

Similar to the Quick Action panel in Detail View 1, this component allows a limited set of Quick Actions that a user is likely to perform from a mobile device. For example, since this template shows an opportunity record, it includes Quick Action icons for taking a note, updating opportunity stage, and marking an opportunity as Closed Won or Closed Lost.

Data Section

Similar to the Data Section in Detail View 1, this component is used to display the selected fields from the respective sObject. For example, this template shows the opportunity name and account associated with an opportunity record.

```
<section class="opportunity-overview">
  <div class="content">
    <h1>$34,100</h1>
    Close date 07/11/2013
  </div>
</section>
```

Additionally, this template also includes two unique visual elements for displaying a picklist field that implies a progression (such as opportunity stage) and a percentage field (probability). These elements can be used in any detail view that displays those two field types.

```
<section>
  <div class="content">
    <div class="progress-dotted" data-completed="4/10">
      <div class="progress-dotted-label">Some Milestone</div>
    </div>

    <div class="progress-bar" data-completed="27">&nbsp;</div>
    <h1 class="inline">27%</h1> probability
  </div>
</section>
```

The template uses some JavaScript and the `data-completed` attribute to display the progression and percentage values.

Related List Section

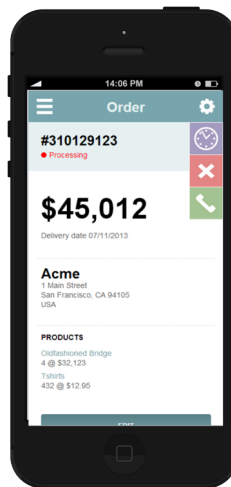
Similar to the Related List Section in Detail View 1, this component can be used to display related child records. For example, this template shows the activity history for the opportunity record.

Each related child record can be made clickable via a `<a>` link and the user can be directed to the respective detail view for that child record on click

Action Buttons

Similar to the action buttons in Detail View 1, this component can be used to display one or more action buttons (like Edit, Delete, etc.).

Detail View 3



In addition to the common [Header](#) component, this template is composed of the following components.

Banner Section

Similar to the banner section in Detail View 1, this component is used to display the most important information related to the record. For example, this template shows the order number and status associated with a custom order record.

```
<div class="detail-view-header with-action-panel">
  <div class="content">
    <h1>#310129123</h1>
    <div class="status status-red"><span class="status-dot">&nbsp;</span>Processing</div>
```

```
</div>
</div>
```

Quick Action Menu

Similar to the Quick Action panel in Detail View 1, this component allows a limited set of Quick Actions that a user is likely to perform from a mobile device. For example, since this template shows an order record, it includes Quick Action icons for showing the order status history (which can be implemented using the [Timeline List View](#)), canceling an order, and contacting support.

Data Section

Similar to the Data Section in Detail View 1, this component is used to display selected fields from the respective sObject. For example, this template shows the order amount and delivery date associated with the order record, the account name, and billing address.

Related List Section

Similar to the Related List Section in Detail View 1, this component can be used to display related child records. For example, it shows the products related to the order record.

Action Buttons

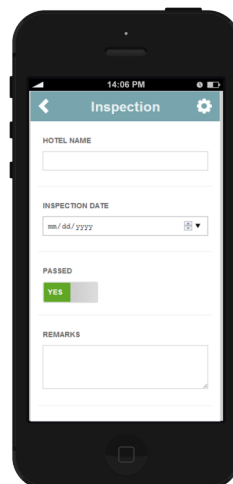
Similar to the Action Buttons in Detail View 1, this component can be used to display one or more action buttons (like Edit, Delete, etc.).

Data Input Templates

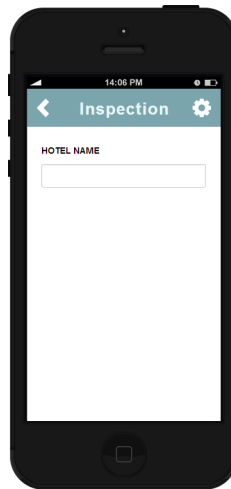
Data input templates use basic HTML5 form elements to capture user input from a phone. The different form elements included in these templates (phone, date, number, text, etc.) can be used in any mobile app that requires users to add or update Salesforce data.

Standard Data Template

The Standard Data Input template includes all the basic HTML5 form elements to capture user input. The InspectionReport Visualforce page of the Mobile Templates Jumpstarter sample app shows an example of this template in action.

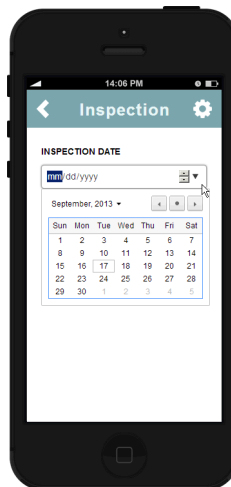


In addition to the common [Header](#) component, this template is composed of the following components.

Text

Add the following markup to accept text input.

```
<section class="border-bottom">
  <div class="content">
    <h3>Hotel Name</h3>
    <div class="form-control-group">
      <div class="form-control form-control-text">
        <input type="text" name="text">
      </div>
    </div>
  </div>
</section>
```

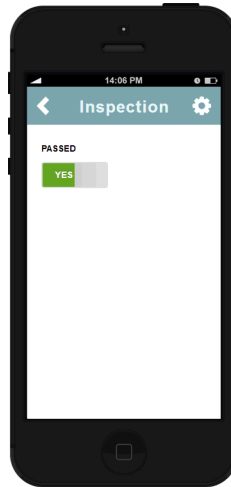
Date

Add the following markup to accept date input.

```
<section class="border-bottom">
  <div class="content">
    <h3>Inspection Date</h3>
    <div class="form-control-group">
      <div class="form-control form-control-date">
        <input type="date" name="date">
      </div>
    </div>
  </div>
```

```
</div>
</section>
```

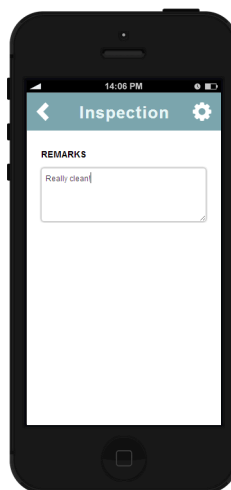
Toggle



Add the following markup to accept a toggle (checkbox) input. Use the `data-on-label` and `data-off-label` attributes to control the text that appears for the on/off positions of the toggle.

```
<section class="border-bottom">
  <div class="content">
    <h3>Passed</h3>
    <div class="form-control-group">
      <div class="form-control form-control-toggle" data-on-label="yes"
data-off-label="no">
        <input type="checkbox" name="toggle">
      </div>
    </div>
  </div>
</section>
```

Textarea



Add the following markup to accept long textarea input.

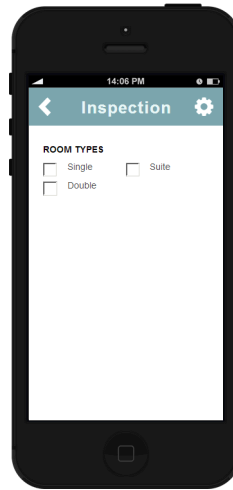
```
<section class="border-bottom">
  <div class="content">
```

```

    <h3>Remarks</h3>
    <div class="form-control-group">
      <div class="form-control form-control-textarea">
        <textarea></textarea>
      </div>
    </div>
  </div>
</section>

```

Multi-select

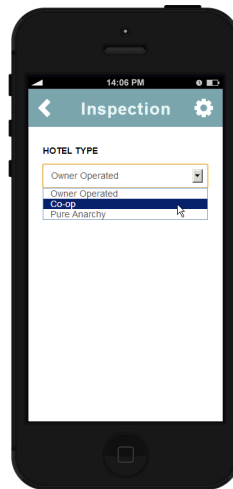


Add the following markup to accept a multi-select picklist value.

```

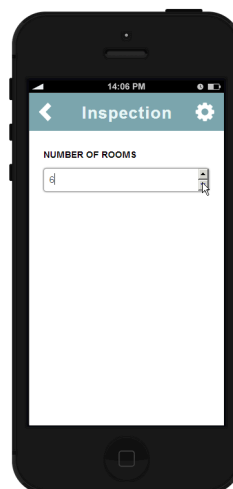
<section class="border-bottom">
  <div class="content">
    <h3>Room Types</h3>
    <div class="span-50 form-control-group">
      <div class="form-control form-control-checkbox">
        <input type="checkbox"><label>Single</label>
      </div>
      <div class="form-control form-control-checkbox">
        <input type="checkbox"><label>Double</label>
      </div>
    </div>
    <div class="span-50 form-control-group">
      <div class="form-control form-control-checkbox">
        <input type="checkbox"><label>Suite</label>
      </div>
    </div>
  </div>
</section>

```

Picklist/Select

Add the following markup to accept a picklist value.

```
<section class="border-bottom">
  <div class="content">
    <h3>Hotel Type</h3>
    <div class="form-control-group">
      <div class="form-control form-control-select">
        <select>
          <option>Owner Operated</option>
          <option>Co-op</option>
          <option>Pure Anarchy</option>
        </select>
      </div>
    </div>
  </div>
</section>
```

Number

Add the following markup to accept a number value.

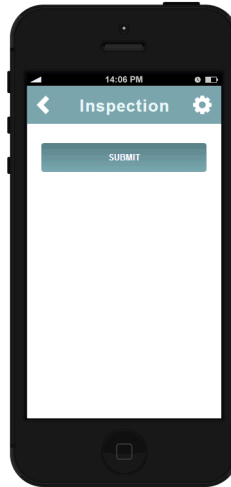
```
<section >
  <div class="content">
    <h3>Number of Rooms</h3>
    <div class="form-control-group">
      <div class="form-control form-control-number">
```

```

        <input type="number" >
      </div>
    </div>
  </div>
</section>

```

Action Buttons



Add the following markup to display one or more action buttons (like Submit, Edit, Delete, etc.).

```

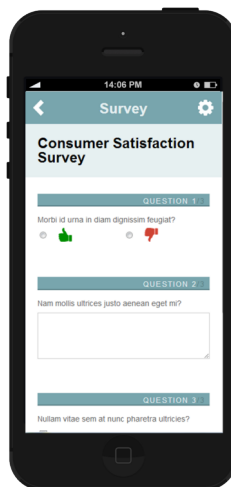
<section class="data-capture-buttons one-buttons">
  <div class="content">
    <a href="#">submit</a>
  </div>
</section>

```

Similarly, the `two-buttons` style can be used to display two buttons in this section.

Survey Data Template

The Survey Data Input template provides a sample design for capturing simple survey responses from a mobile device. The Take_Survey Visualforce page of the Mobile Templates Jumpstarter sample app shows an example of this template in action.



All the form elements used in this template are common to the previous Standard Data Input template, except for the following two components.

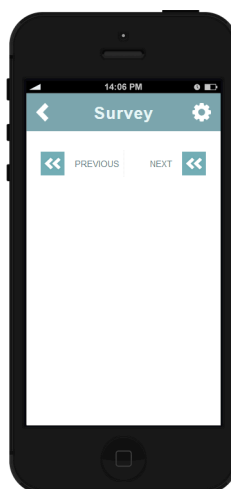
Toggle



Similar to the toggle control used in the Standard Data Input template, the following markup allows users to provide yes/no type binary input:

```
<section>
  <div class="content">
    <form>
      <div class="form-header-survey">Question 1<span
class="color-menu-blue">/4</span></div>
      <p>Morbi id urna in diam dignissim feugiat?</p>
      <div class="form-control-group">
        <div class="form-control form-control-radio form-control-radio-thumbs
span-50">
          <input type="radio" name="name" value="yes"><label class="thumbs
thumbs-up">&nbsp;</label>
        </div>
        <div class="form-control form-control-radio form-control-radio-thumbs
span-50">
          <input type="radio" name="name" value="no"><label class="thumbs
thumbs-down">&nbsp;</label>
        </div>
      </div>
    </form>
  </div>
</section>
```

Navigation Buttons



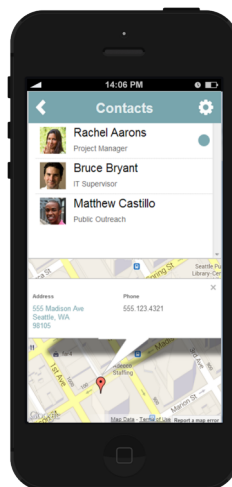
The Previous and Next navigation buttons provide an alternative styling to the action buttons used in the Standard Data Input template.

```
<section class="form-navigation-buttons">
  <div class="content">
    <a href="#" class="span-50 previous">previous</a>
    <a href="#" class="span-50 next">next</a>
  </div>
</section>
```

In addition to the Previous and Next navigation buttons, a `finish` style class can be used to display a Finish button in the Survey template.

Map View Templates

Map view templates provide Google Map-based designs for implementing the common 'Find Nearby' functionality on a mobile device. These templates can be combined with the `Geolocation` custom field in Salesforce and its corresponding SOQL companion to add geolocation functionality to any Web or hybrid mobile app.



Map View 1

This template displays a list of data records (contacts in this example) on the top half of the page, and the location of any selected data record on the bottom half of the page. Developers can use the HTML5 geolocation feature and a SOQL distance query to find nearby data to display on the top half of the page. When a user selects any record from the list, the corresponding latitude and longitude coordinates are displayed in the Google Map on the bottom half of the page. A user can also click the selected address in the Google Map to launch the native Map application (iOS or Android) on the device.

In addition to the common Header component, this template is composed of the following components.

List

The list section displays the data set that is typically queried from Salesforce.

```
<ul class="list-view left-thumbs right-one-icons">
  <li>
    <div class="thumbs">
      
    </div>
  </li>
</ul>
```

```

        <a href="#" class="content" data-map-id="0">
          <h2>Rachel Aarons</h2>
          Project Manager
          <div class="list-view-icons">
            <span class="icon-map-dot">&nbsp;</span>
          </div>
        </a>
      </li>
</ul>

```

Map

Clicking any record in the list section shows a marker on the Google Map for the corresponding latitude and longitude coordinates.

```

<div id="map-canvas-wrapper" class="map-canvas-wrapper map-canvas-wrapper-list-view">
</div>

```

This Google Map interaction is implemented in the `main.js` script file. Some minor JavaScript initialization code is required in order to pass the latitude and longitude coordinates for each of the records displayed in the list section to the `mapObj` object in `main.js`.

```

var markers = [
  {
    id: 0,
    lat: 47.604789,
    lng: -122.335682,
    contentString: "<div class='infowindow-container'><p
class='span-50'><strong>Address</strong><br/><a
href='https://maps.apple.com/maps?q=47.620513,-122.34861'>555 Madison Ave<br/>Seattle,
WA<br/>98105</a></p><p class='span-50'><strong>Phone</strong><br/>555.123.4321</p></div>"
  }
];

mapObj.init({
  view: "listView",
  markers: markers
});

```

Note the `maps.apple.com` link passed to the `contentString` variable. If viewed on an iPhone, this will launch the native iOS Map application when the user clicks the corresponding address on the Google Map (for directions, etc.). A different URL can be used to achieve the equivalent functionality on an Android device.

Map View 2

This template provides an alternative implementation for the 'Find Nearby' use case. The page shows the current location of the device on a Google Map (using the HTML5 geolocation feature), as well as a set of nearby data points. As with the previous template, these nearby data records will typically be retrieved via a SOQL distance query. A user can also click any marker on the Google Map to launch the native Map application (iOS or Android) on the device.

In addition to the common Header component, this template is composed of the following components.

Map

A simple container `<div>` tag is required to host the Google Map on the page.

```

<div id="map-canvas-wrapper">&nbsp;</div>

```

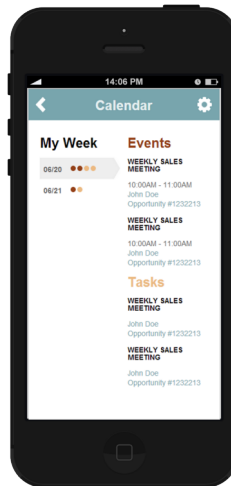
This Google Map interaction is implemented in the main.js script file. Some minor JavaScript initialization code is required in order to pass the latitude and longitude coordinates for each of the records displayed in the list section to the mapObj object in main.js.

```
var markers = [
  {
    id: 0,
    lat: 47.604789,
    lng: -122.335682,
    contentString: "<div class='infowindow-container'><p
class='span-50'><strong>Address</strong><br/><a
href='https://maps.apple.com/maps?q=47.620513,-122.34861'>555 Madison Ave<br/>Seattle,
WA<br/>98105</a></p><p class='span-50'><strong>Phone</strong><br/>555.123.4321</p></div>"
  }
];

mapObj.init({
  view: "listView",
  markers: markers
});
```

Calendar View Templates

Calendar View templates provide mobile optimized views of a user's Salesforce calendar (tasks and events).

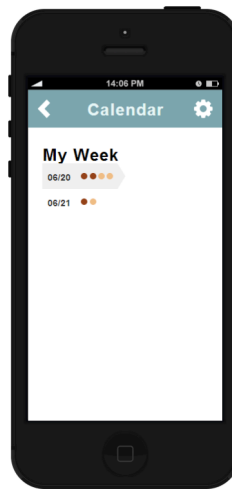


Day Calendar

This template provides a mobile-optimized view of a user's scheduled tasks and events in Salesforce.

In addition to the common Header component, this template is composed of the following components.

Week View



This component displays the current week on the left half of the page. Each scheduled task is displayed as a dark brown dot, and each event as a light brown dot for any given day of the week.

```
<div class="span-50 padding-right-gutter-half">
  <h1 class="padding-bottom-gutter">My Week</h1>
  <ul class="week-planner">
    <li data-date="06-20">
      <div class="date">06/20</div>
      <ul class="week-planner-items">
        <li class="event"></li>
        <li class="event"></li>
        <li class="task"></li>
        <li class="task"></li>
      </ul>
    </li>
  </ul>
</div>
```

Note the use of the week-planner-items, event, and task style classes in the and tags to display the list of tasks and events as dots.

Calendar Entries

This component displays the detailed list of tasks and events on the right half of the page when a user selects a particular date in the week view.

```
<div class="span-50 padding-left-gutter-half">
  <div class="date-content" id="date-content-06-20">
    <h1 class="event">Events</h1>
    <div class="events">
      <h3>Weekly Sales meeting</h3>
      <p>
        10:00AM - 11:00AM<br/>
        <a href="#">John Doe</a><br/>
        <a href="#">Opportunity #1232213</a>
      </p>
    </div>
    <h1 class="task">Tasks</h1>
    <div class="tasks">
      <h3>Weekly Sales meeting</h3>
      <p>
        <a href="#">John Doe</a><br/>
        <a href="#">Opportunity #1232213</a>
      </p>
    </div>
  </div>
</div>
```

```

        </div>
    </div>
</div>

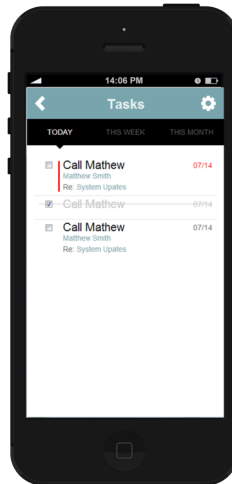
```

Note the use of the `tasks` and `events` style classes to display the tasks and events in different colors on the page. The `<a>` links included in this section can be used to navigate a user to the detail view of the respective task or event record.

The logic to display the correct set of tasks and events to display for a given day is encapsulated in the `main.js` file.

To-do Tasks

This template provides a mobile optimized view of a user's scheduled tasks and events in Salesforce.



In addition to the common [Header](#) component, this template is composed of the following components.

Tab

Tab is used to group a user's pending tasks by Today, This Week, and This Month.

```

<div id="tabbed-list-view-nav" class="tabbed-list-view-nav">
  <a href="#" class="span-33 on">Today</a>
  <a href="#" class="span-33">This Week</a>
  <a href="#" class="span-33">This Month</a>
  <div id="tabbed-list-view-nav-arrow" class="tabbed-list-view-nav-arrow">&nbsp;</div>
</div>

```



Note: Note that this is the same tab component used in the [Tabbed List View](#) template.

Tasks

Tasks displays a list of pending tasks. Each task can be added to the list using the following markup.

```

<ul class="list-view list-view-tasks">
  <li>
    <div class="content">
      <div class="task-complete-checkbox">
        <input type="checkbox" name="checkbox" />
      </div>
      <div class="date">07/14</div>
      <h2>Call Mathew</h2>
      <p><a href="#">Matthew Smith</a></p>
    </div>
  </li>
</ul>

```

```

        <p>Re: <a href="#">System Updates</a></p>
      </div>
    </li>
  </ul>

```

Overdue Tasks

To mark a task as overdue, add the `overdue` style class to the `` tag.

```
<li class="overdue">
```

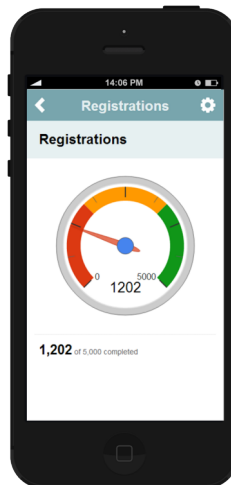
Completing a Task

This template uses some JavaScript in the `main.js` file to achieve the strikethrough effect when a user clicks the checkbox to complete a task.

Report and Dashboard Templates

Dashboard templates provide mobile optimized report views of Salesforce data. These templates are developed using the open-source D3 charting library and developers can combine them with the Salesforce Analytics API to add reporting capabilities to their mobile apps. The InventoryReport Visualforce page of the Mobile Templates Jumpstarter sample app shows an example of using these templates to display mobile optimized reports.

Gauge



The Gauge template uses the D3 Gauge Chart to provide a gauge view of a Salesforce report.

```

<section>
  <div class="content">
    <div id="graph" class="graph"></div>
  </div>
</section>

```

The Gauge chart is currently initialized with some hard-coded data:

```

$(document).ready(function() {
  reportObj.init({
    graphType: "gauge",
    width: 300,
    gauge: {

```

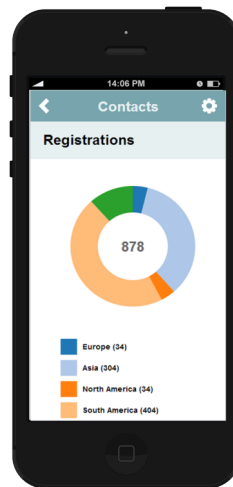
```

        max: 5000,
        value: 1202
      }
    });
  });
};

```

You should replace this hard-coded data with live Salesforce reporting data using a call to the Analytics API. You can also peruse the `main.js` script for the JavaScript logic used to render the D3 Gauge chart.

Donut



The Donut template uses the D3 Donut Chart to provide a distribution view of Salesforce data.

```

<section>
  <div class="content">
    <div id="graph" class="graph"></div>
  </div>
</section>

```

The donut chart is currently initialized with some hard-coded JSON data.

```

$(document).ready(function() {
  reportObj.init({
    graphType: "donut",
    jsonURL: "json/survey.json"
  });
});

```

Replace this hard-coded data with live Salesforce reporting data via a call to the Analytics API. Developers can also peruse the `main.js` script for the JavaScript logic used to render the D3 Donut chart.

Bar Chart



The Bar Chart template is based on the D3 Bar Chart.

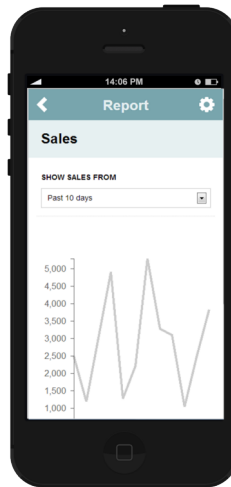
```
<section>
  <div class="content">
    <div id="graph" class="graph"></div>
  </div>
</section>
```

The bar chart is currently initialized with some hard-coded JSON data.

```
$(document).ready(function() {
  reportObj.init({
    graphType: "bar",
    jsonURL: "json/produts.json",
    axis : {
      xAxisIndex: "name",
      yAxisIndex: "inventory"
    }
  });
});
```

Replace this hard-coded-data with live Salesforce reporting data via a call to the Analytics API. You can also peruse the `main.js` script for the JavaScript logic used to render the D3 Bar Chart.

Line Chart



The Line Chart template is based on the D3 Line Chart.

```
<section>
  <div class="content">
    <div id="graph" class="graph"></div>
  </div>
</section>
```

The line chart is currently initialized with some hard-coded JSON data.

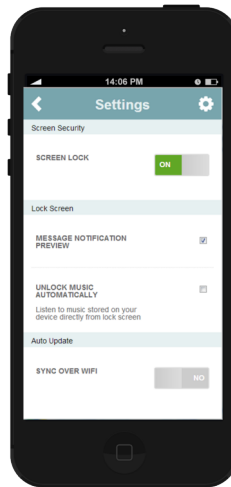
```
$(document).ready(function() {
  reportObj.init({
    graphType: "line",
    jsonURL: "json/sales.json",
    line: {
      periods : [10, 30]
    },
    axis: {
      xAxisIndex: "date",
      yAxisIndex: "total"
    }
  });
});
```

Replace this hard-coded data with live Salesforce reporting data via a call to the Analytics API. Developers can also peruse the `main.js` script for the JavaScript logic used to render the D3 Line Chart.

Miscellaneous Templates

This category of templates provides simple designs for functionality common to most mobile apps. Developers can take these templates as a starting point to add a settings, splash or about screen to their mobile app.

Settings Screen



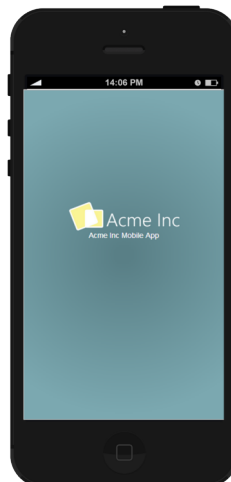
Use this template to allow a user to view and update the settings associated with your mobile app.

```
<div class="list-view-header">Screen Security</div>
  <section>
    <div class="content">
      <div class="span-66 settings settings-left">
        <h3>Screen Lock</h3>
      </div>
      <div class="span-33 settings settings-right">
        <div class="form-control-group">
          <div class="form-control form-control-toggle" data-on-label="on"
data-off-label="off">
            <input type="checkbox" name="toggle">
          </div>
        </div>
      </div>
    </div>
  </div>
</section>
```



Note: The form elements used in this template are the same as those in the [Standard Data Input](#) template.

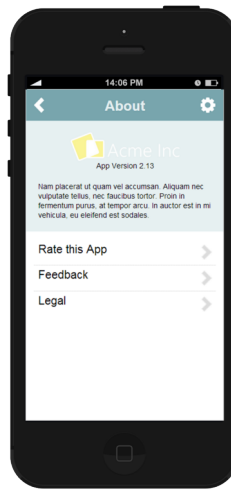
Splash Screen



Use this template to show an initial custom, branded splash screen when a user first launches the mobile app.

```
<div class="splash-screen-wrapper">
  <div class="content">
    <div class="logo">
      
    </div>
    <div class="tagline">
      Acme Inc Mobile App
    </div>
  </div>
</div>
```

About Screen



Use this template to show an about screen with credits for the mobile app.

```
<div class="detail-view-header detail-view-header-about">
  <div class="content">
    
    <div class="version">App Version 2.13</div>
    <div class="description">Some Description</div>
  </div>
</div>

<ul class="list-view right-one-icons padding-top-gutter">
  <li>
    <a href="#" class="content">
      <h2>Rate this App</h2>
      <div class="list-view-icons">
        <span class="icon-right-arrow">&nbsp;</span>
      </div>
    </a>
  </li>
</ul>
```

Mobile Packs

Salesforce Mobile Packs let you build web and hybrid apps that integrate with the Salesforce1 Platform using industry-standard web technologies like HTML5, CSS3 and JavaScript. The number of supported Mobile Packs is continually growing. The following are currently available:

- JavaScript/HTML5 coding frameworks:

- ◇ Backbone.js is a JavaScript framework that provides models with key-value binding and custom events, collections with a rich API of enumerable functions, and views with declarative event handling.
 - ◇ Google's AngularJS lets you reap the benefits of a Model-View-Control architecture in your JavaScript code and utilize advanced features like reusable components and dependency injection.
 - ◇ jQuery Mobile is a touch-optimized web framework that lets you develop mobile web applications using HTML5, JavaScript and CSS3 for a wide variety of smart phones and tablet computers.
 - ◇ Knockout adds dependency tracking that refreshes your UI whenever your data changes and lets you create new reusable custom behaviors and responsive designs.
 - ◇ Sencha leverages HTML5 components with built-in state management and fluid animations to display customer data in visually dynamic ways.
- Powerful cross-platform build and development utilities:
 - ◇ Appery.io helps you build apps using a cloud-based, visual drag-and-drop editor that easily connects to any Salesforce APIs. Includes Contacts and OAuth sample apps that you can reuse.
 - ◇ Codiqa helps you rapidly prototype mobile apps that connect to Salesforce using pure HTML5. Leverage open-source HTML5 components to go from idea to deployment and use Salesforce Mobile Services to securely connect to trusted cloud data.
 - ◇ Xamarin accesses Salesforce1 Platform features like security and identity, and uses declarative bindings to invoke third-party Java, Objective-C, and C++ libraries and call them directly from C#. Can be used to create a bridge between .NET apps and Salesforce.

For the most current information on mobile packs, see <http://www2.developerforce.com/mobile/services/mobile-packs>.

jQuery Quick Start

jQuery Mobile is a touch-optimized Web framework that lets you develop mobile Web applications using HTML5, JavaScript, and CSS3 for a wide variety of smart phones and tablet computers. In this quick-start tutorial, you'll learn how to combine jQuery Mobile with the Salesforce Web framework, Visualforce, natively on Force.com.

This quick start creates a contact management app that provides basic CRUD access to the contact standard object. Using this app, users can view, update, add, and delete contact records from any mobile device with a modern browser. Here's the completed application.



Install the jQuery Mobile Pack

To install the Mobile Pack for jQuery Mobile into your DE Org:

1. Log into your DE org.
2. Paste this install URL into the address bar (or click the link) to start installing the package:
<https://login.salesforce.com/packaging/installPackage.apexp?p0=04ti0000000JElkf>
3. Follow the package installation instructions.

What's In the Mobile Pack?

- **MobileSample_jQueryMobile.page / jQuery Mobile Tab**—This Visualforce page forms the heart of the application and contains both the list view and the update/add/delete view shown above. You can navigate to your page by manually redirecting in the URL or by using the included jQuery Mobile tab.
- **MobileSample_Resources_jQueryMobile (Static Resource zipfile)**—This package contains all the CSS files, images, and JS libraries used in the quick start.
- **RemoteTK VF component, Apex class, test class**—Unlike the other mobile packs, this mobile pack uses RemoteTK which provides an abstraction very similar to the REST API, implemented via @RemoteAction methods in the controller. However, it doesn't consume any API calls. Check out the Force.com JavaScript REST Toolkit README to learn more about how the custom component works.

Test the App

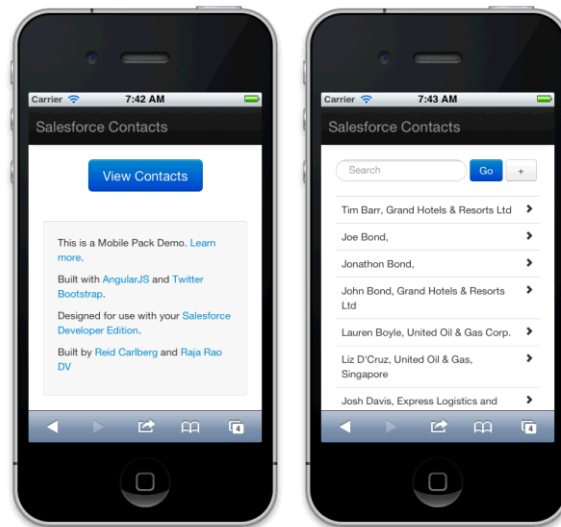
1. Navigate to the MobileSample_jQueryMobile Visualforce page by appending the page name to the end of your instance URL—https://login.salesforce.com/apex/MobileSample_jQueryMobile.
2. You should see the application working.
3. Test the page from a mobile device (after logging in with your Salesforce credentials).
4. To modify the page or view the code, in Setup, click **Develop > Pages**.

Next Steps

- The source files and latest updates are on Github—<https://github.com/developerforce/MobilePack-jQueryMobile>
- Salesforce Mobile Packs are evolving quickly, so you'll want to check back frequently for the latest enhancements and sample apps: <http://www2.developerforce.com/en/mobile/getting-started/html5/#jquery>

Angular.js Quick Start

Angular.js is a JavaScript framework that dramatically improves the Web app authoring experience. This sample app uses Angular.js to create a cross-platform Web app to access your Salesforce data using the REST API. It also uses Twitter Bootstrap responsive design, all while running inside a Visualforce page. Here's the completed application.



Install the Mobile Pack

To install the Mobile Pack into your DE Org:

1. Log into your DE org.
2. Paste this install URL into the address bar (or click the link) to start installing the package:
<https://login.salesforce.com/packaging/installPackage.apexp?p0=04ti0000000Vc60>
3. Follow the package installation instructions.

What's In the Mobile Pack?

- **MobileSample_ngIndex.page**—This Visualforce page is the starting point for the sample app. The sample app has a total of six Visualforce pages with functionality pertaining to viewing the list of contacts, adding a contact, and viewing/editing/deleting an individual contact.
- **MobileSample_Resources_Angular (Static Resource zipfile)**—This package contains all the CSS files, images, and JavaScript libraries used in the quick start. The application uses angular-force.js to serve dynamic content through two-way data-binding that allows for the automatic synchronization of models and views.

Test the App

1. Navigate to the MobileSample_Angular Visualforce page by appending the page name to the end of your instance URL - https://login.salesforce.com/apex/MobileSample_Angular.
2. You should see the application working.
3. Test the page from a mobile device (after logging in with your Salesforce credentials).
4. To modify the page or view the code, in Setup, click **Develop** > **Pages**.

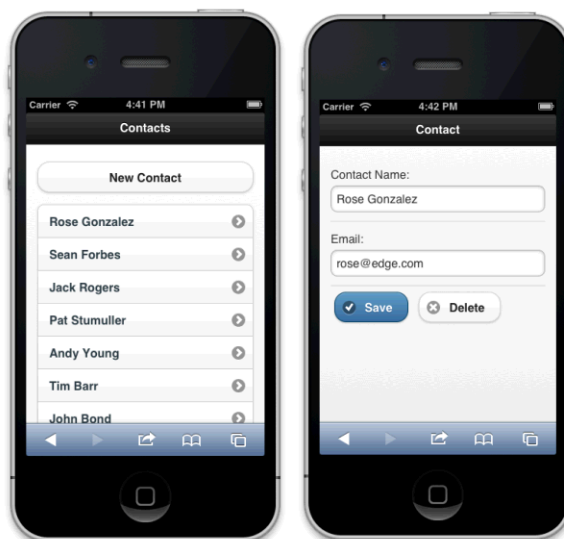
Next Steps

- The source files and latest updates are on Github—<https://github.com/developerforce/MobilePack-AngularJS>
- Salesforce Mobile Packs are evolving quickly, so you'll want to check back frequently for the latest enhancements and sample apps: <https://github.com/developerforce/MobilePack-AngularJS>

Backbone.js Quick Start

Backbone.js provides a structure for JavaScript-heavy applications by providing models with key-value binding and custom events, collections with a rich API of enumerable functions, and views with declarative event handling, while connecting it all to your existing application over a RESTful JSON interface. In this quick start tutorial, you'll learn how to combine Backbone.js with the Salesforce Web framework, Visualforce, natively on Force.com.

This quick start creates a contact management app that provides basic CRUD access to the contact standard object. Using this app, users can view, update, add, and delete contact records from any mobile device with a modern browser. Here's the completed application.



Install the Mobile Pack

To install the Mobile Pack into your DE Org:

1. Log into your DE org.
2. Paste this install URL into the address bar (or click the link) to start installing the package:
<https://login.salesforce.com/packaging/installPackage.apexp?p0=04ti0000000W78j>
3. Follow the package installation instructions.

What's In the Mobile Pack?

- **MobileSample_Backbone.page**—This Visualforce page forms the heart of the application and contains both the list view and the update/add/delete view shown above.
- **MobileSample_Resources_Backbone (Static Resource zipfile)**—This package contains all the CSS files, images and JS libraries used in the quick start. The application uses backbone.js for the MVC logic, and forcetk.js (a JavaScript wrapper for the Force.com REST API) to grab the data from Salesforce. It also uses jQuery Mobile 1.3.0 and jQuery 1.9.1 for the front-end UI.

Test the App

1. Navigate to the MobileSample_Backbone Visualforce page by appending the page name to the end of your instance URL - https://login.salesforce.com/apex/MobileSample_ngIndex.Backbone
2. You should see the application working.
3. Test the page from a mobile device (after logging in with your Salesforce credentials).

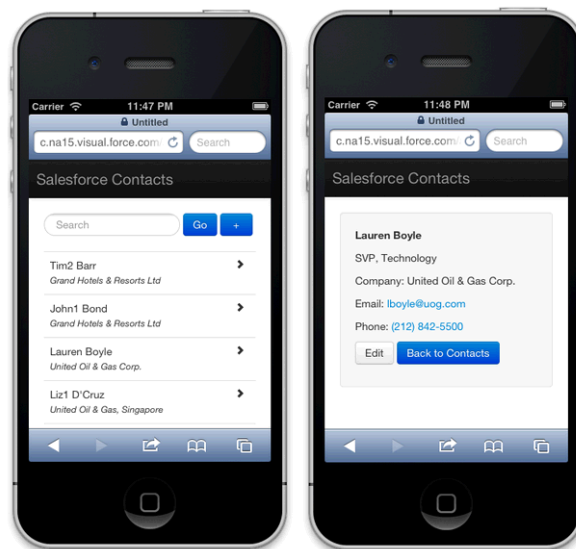
4. To modify the page or view the code, in Setup, click **Develop** > **Pages**.

Next Steps

- The source files and latest updates are on Github—<https://github.com/developerforce/MobilePack-BackboneJS>
- Salesforce Mobile Packs are evolving quickly, so you'll want to check back frequently for the latest enhancements and sample apps: <http://www2.developerforce.com/en/mobile/getting-started/html5/#backbone>

Knockout Quick Start

Knockout is a JavaScript framework that dramatically improves the Web app authoring experience. This quick start creates a contact management app that provides basic CRUD access to the contact standard object. Using this app, users can view, update, add, and delete contact records from any mobile device with a modern browser. This sample app uses Knockout and Twitter Bootstrap responsive design. Here's the completed application.



Install the Mobile Pack

To install the Mobile Pack into your DE Org:

1. Log into your DE org.
2. Paste this install URL into the address bar (or click the link) to start installing the package:
<https://login.salesforce.com/packaging/installPackage.apexp?p0=04ti0000000Vrcj>
3. Follow the package installation instructions.

What's In the Mobile Pack?

- **MobileSample_koIndex.page**—This Visualforce page is the starting point for the sample app. The app has a total of six Visualforce pages with functionality pertaining to viewing the list of contacts, adding a contact, and viewing/editing/deleting an individual contact.
- **MobileSample_Resources_Knockout (Static Resource zipfile)**—This package contains all the CSS files, images, and JS libraries used in the quick start. The application uses `knockout-force.js` to serve dynamic content through two-way data-binding that allows for the automatic synchronization of models and views. It also uses jQuery Mobile 1.3.0 and jQuery 1.9.1 for the front-end UI.

Test the App

1. Navigate to the `MobileSample_koIndex` Visualforce page by appending the page name to the end of your instance URL - `https://login.salesforce.com/apex/MobileSample_koIndex`
2. You should see the application working.
3. Test the page from a mobile device (after logging in with your Salesforce credentials).
4. To modify the page or view the code, in Setup, click **Develop** > **Pages**.

Next Steps

- The source files and latest updates are on Github—<https://github.com/developerforce/MobilePack-KnockoutJS>
- Salesforce Mobile Packs are evolving quickly, so you'll want to check back frequently for the latest enhancements and sample apps: <http://www2.developerforce.com/en/mobile/getting-started/html5/#knockout>

Mobile UI Elements

Mobile apps should be small, fun, and engaging. Mobile app developers should spend their time creating innovative functionality, rather than re-creating yet another list view or detail page bound to a set of APIs. With Salesforce Mobile UI Elements, HTML and JavaScript developers can build amazing apps with technologies they already know—using a set of pre-built components that are flexible and surprisingly easy to learn.

You can deploy a Mobile UI Elements app several ways.

- In a Visualforce page
- Hosted as a multi-dimensional publisher (MDP) action
- As a stand-alone app, using the hybrid container provided by Salesforce Mobile SDK

Mobile UI Elements is an open-source framework based on Google's Polymer framework. It provides fundamental building blocks that you can combine to create fairly complex mobile apps. The component library enables any HTML developer to quickly and easily build mobile applications without having to dig into complex mobile frameworks and design patterns.

You can find the source code for Mobile UI Elements on Github at <http://bit.ly/mobile-ui-elements>.

Available UI Elements

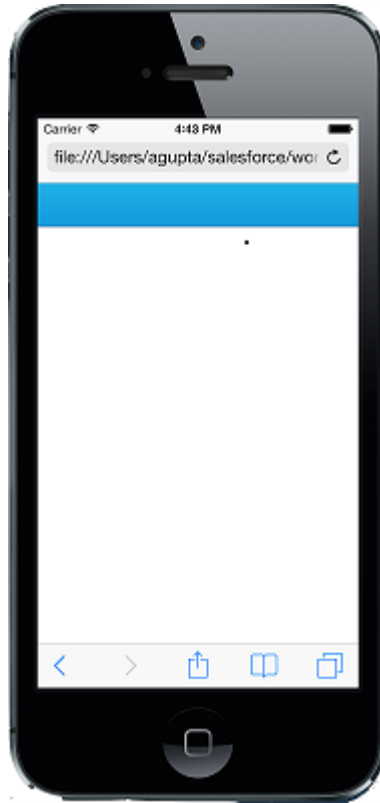
force-ui-app

A top-level UI element that provides the basic styling and structure to the application. This element also brings in all the common JavaScript libraries required for other UI elements. This element is also responsible for managing the session ID, via `accessToken` attribute, for all the API calls. Supported attributes include:

- `accessToken`
- `instanceurl`
- `multipage`
- `startpage`
- `hideheader`

For example, to use `force-ui-app` inside Visualforce:

```
<force-ui-app accessToken="{!$Api.Session_ID}"></force-ui-app>
```

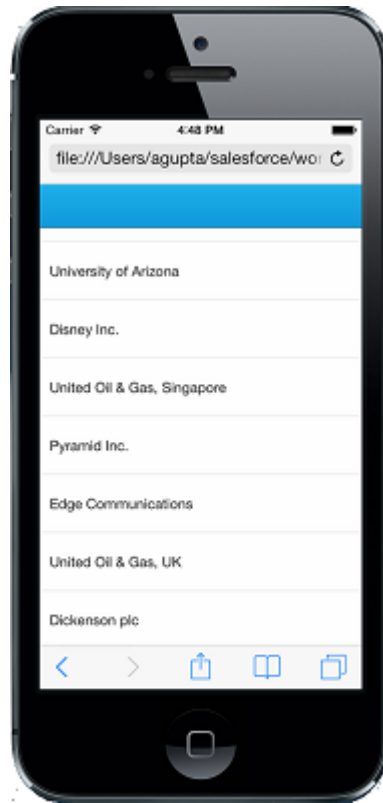
**force-ui-list**

Enables rendering of a list of records for any standard object. You configure this element's attributes to show a specific set of records. Supported attributes include:

- `subject`
- `query`
- `querytype`

This element is intended to be a child of the `force-ui-app` element. For example:

```
<force-ui-app accesstoken="{!$Api.Session_ID}">
  <force-ui-list subject="Account" querytype="mru"></force-ui-list>
</force-ui-app>
```



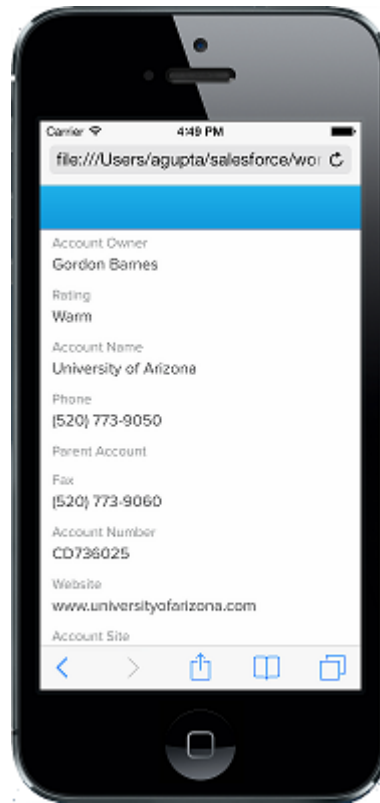
force-ui-detail

Provides a quick and easy way to render a full view of a Salesforce record. This element auto-detects the record's relevant page layout and renders the details appropriately. The element can be configured by using the various attributes, such as `subject`, `recordid` etc, to render layout of a particular record. This element should always be a child of `force-ui-app` element. Supported attributes include:

- `subject`
- `recordid`
- `hasrecordtypes`
- `recordtypeid`

For example:

```
<force-ui-app accesstoken="{!$Api.Session_ID}">
  <force-ui-detail subject="Account" recordid="001000000000AAA"></force-ui-detail>
</force-ui-app>
```



Third-Party Code

The Mobile UI Elements library makes use of these third-party components:

- [Polymer](#), a JavaScript library for adding new extensions and features to modern HTML5 browsers. It's built on Web Components and is designed to leverage the evolving Web platform on modern browsers.
- [jQuery](#), the JavaScript library that makes it easy to write JavaScript.
- [Backbone.js](#), a JavaScript library providing the model-view-presenter (MVP) application design paradigm.
- [Underscore.js](#), a “utility belt” library for JavaScript.
- [Ratchet](#), prototype iPhone apps with simple HTML, CSS, and JavaScript components.

Using the Camera in HTML5: Mobile UI Elements Sample App

The Mobile UI Elements sample application shows how to use the `force-ui-app`, `force-ui-list` and `force-ui-detail` elements in an Apex page to render a list of account records. From the list, the user can navigate to another page to view the full details of a selected account. This application also demonstrates how to use an HTML5 `<input>` element with the SmartSync API to launch the device camera application, take a picture, and attach it to the account record. You can install the sample Visualforce page in your Developer Edition org from <http://bit.ly/mobile-vf-package>.

This simple app performs three basic tasks.

- Loads the necessary stylesheet and libraries:
 - ◇ `ratchet.css`—For flexible styling.
 - ◇ `polymer.min.js`—The Polymer framework from Google is the underlying technology for Mobile UI Elements.
 - ◇ `mobile-ui-elements.html`—The Mobile UI Elements library.
- Defines a Polymer element named `mobile-app` that comprises two parts:

- ◊ A layout template that defines the UI design. This is where the Mobile UI Elements come into play.
- ◊ Defines a Polymer script for the `mobile-app` element. The script consists of a JSON object with event listeners that handle navigation to the detail screen and camera input.
- Instantiates a `mobile-app` instance within the HTML page.

Let's examine the code for each of these steps. First, there's a `<head>` element for the libraries and CSS imports:

```
<apex:page docType="html-5.0" showHeader="false"
  sidebar="false" standardStylesheets="false">
<html>
  <head>
    <!-- viewport meta tag to set the width of the page as
      the width of the device screen -->
    <meta name="viewport" content="width=device-width"/>
    <!-- Importing the ratchet stylesheet
      for styling the app -->
    <link rel="stylesheet"
      href="{!URLFOR($Resource.MobileUIElements,
        'mobile-ui-elements/css/ratchet.css')}" />
    <!-- Importing the polymer javascript library -->
    <script src="{!URLFOR($Resource.MobileUIElements,
      'dependencies/polymer/polymer.min.js')}"></script>

    <!-- Importing all the force.com Mobile UI Elements
      using HTML imports -->
    <link rel="import"
      href="{!URLFOR($Resource.MobileUIElements,
        'mobile-ui-elements/mobile-ui-elements.html')}" />
  </head>
```

This part is very straightforward. Let's move on to the Polymer element definition.

The `mobile-app` element definition occurs within the HTML `<body>` element. Within the `<polymer-element>` node lies the definition of the entire application. Defining the application as a custom Polymer element helps you create custom templates and also allows dynamic data binding between different elements.

```
<body>
  <polymer-element name="mobile-app">
    <template>
      <force-ui-app id="force_ui_app" multipage="true"
        accesstoken="{!$Api.Session_ID}">
        <force-ui-list id="force_ui_list"
          subject="Account" class="page content"
          on-polymer-activate="showDetail">
        </force-ui-list>
        <force-ui-detail id="force_ui_detail"
          recordid="{!$.force_ui_list.selected}"
          class="page content">
          <div id="attachments"
            style="margin:10px"></div>
          <a class="button" style="margin:10px">
            Attach Photo
            <input type="file" on-change="attach"
              style="opacity: 0;
              position: absolute; top: 0;
              left: 0;" />
          </a>
        </force-ui-detail>
      </force-ui-app>
    </template>
    ...
```

This template uses three Mobile UI Elements:

force-ui-app

Provides a multipage structure to the application. Also sets up the access token for API calls. Uses the following attributes:

id

The unique ID assigned to this element (in this case, `force_ui_app`.) This ID allows other elements to reference this element using the following syntax: `$.force_ui_app`.

multipage

If true, Polymer distributes each child element to an individual page for full view. Also allows navigation between these pages.

access token

The token that allows UI elements to call APIs on the user's behalf. In this case, `{!$Api.Session_ID}` is the Visualforce API to get the session token of the current user.

force-ui-list

Embedding the `force-ui-list` component to render the list of most recently used account records. Uses the following attributes:

id

The unique ID assigned to this element (in this case, `force_ui_list`.) This ID allows other elements to reference this element using the following syntax: `$.force_ui_list`

subject

Salesforce object from which records are fetched (in this case, `accounts`.)

class

Name of class that determines how the app renders this element (in this case, `"page"`.) This sample uses the following style classes:

- `"page"`—Tells `force-ui-app` to show this list element as a full page view
- `"content"`—Adds Ratchet styling to allow scrolling on the list

force-ui-detail

Embedding the `force-ui-detail` component to render the detail of an account record. Uses the following attributes:

id

The unique ID assigned to this element (in this case, `force_ui_detail`.) This ID allows other elements to reference this element using the following syntax: `$.force_ui_detail`

recordid

ID of the account record whose details are displayed (in this case, `{{$.force_ui_list.selected}}`.) `"{{{}}`" syntax allows dynamic assignment of values between different elements. In this case, `$.force_ui_list.selected` returns the ID of the selected record in the list view. That ID is transparently assigned to the `force-ui-detail` element to fetch the full record details.

class

Name of class that determines how the app renders this element (in this case, `"page"`.) This sample uses the following style classes:

- `"page"`—Tells `force-ui-app` to show this list element as a full page view

- "content"—Adds Ratchet styling to allow scrolling on the list

Of these three components, `force-ui-detail` is most interesting because it contains the HTML code that launches the camera:

```
<input type="file" on-change="attach" style="opacity: 0;
      position: absolute; top: 0; left: 0;"/>
```

The HTML5 `<input>` node, when used on a mobile device with the `type` attribute set to `file`, searches for a device camera. If it finds one, it launches the camera application. If no camera is available, it prompts the user to choose an image file instead. This element also specifies a function named `attach` as the handler for the `<input>` node's `on-change` event (meaning that the user has clicked the camera shutter or chosen a file).

The script that defines the `mobile-app` functionality lives in a script tag in the `<body>` element. It begins by defining a Polymer symbol named `mobile-app` and assigning to that symbol a JSON string that implements the app's functionality. There are two methods:

- `showDetail(e)`—Listener for the `on-polymer-activate` event of the `force-ui-list` element.
- `attach(e)`—Listener for the `on-change` event of the HTML5 `<input>` element.

The `showDetail()` method is responsible for asking Polymer to navigate to the `force-ui-detail` screen when the user clicks on an item in the `force-ui-list` screen. Mobile UI Elements handles the transfer of the selected record ID to the detail screen transparently, without any need for special coding in the app.

The `attach()` method responds to the camera click (or the user's file selection if the user doesn't take a picture) by posting the file to the account record on the server. This is handled in the `saveFileToSFDC()` function. The `attach()` method creates a standard `FileReader` JavaScript object and calls the `saveFileToSFDC()` function from that object's `onloadend` event listener. To save the file to the Salesforce server, `saveFileToSFDC()` uses an object of the `SmartSync Force.SObject` class.

```
<script type="text/javascript">
  // Defining the new element with name "mobile-app"
  // and a JSON object with Javascript methods
  // "showDetail" and "attach" that will listen
  // to events "polymer-activate" and
  // "change" respectively.
  Polymer('mobile-app', {
    // This method is assigned on force-ui-list
    // element as the listener to
    // "polymer-activate" event.
    // @params:
    //   e: Event object for the "polymer-activate"
    //   event.
    showDetail: function(e) {
      // Check if a record is selected and not
      // de-selected. Then use the "navigateTo"
      // method on "force-ui-app" element
      // (id: force_ui_app) to slide to
      // "force-ui-detail" element
      // (id: force_ui_detail).
      this.$.force_ui_app.navigateTo(
        '#force_ui_detail');
    },
    // This method is assigned on input[type=file]
    // element as the listener to "change" event.
    // @params:
    //   e: Event object for the "change" event.
    attach: function(e) {
      var that = this;
```



```

// Defining the method "saveFileToSFDC" to
// save base64 encoded file contents to
// Salesforce as an attachment to an Account.
// @param:
// blob: base64 encoded string of file
// contents.
var saveFileToSFDC = function(blob) {
  // Create an instance of Force.SObject
  // provided by Smartsync.
  // Smartsync comes packaged as part of
  // Mobile UI Elements.
  var attachment = new Force.SObject();
  // Setting the objectType property as
  // "Attachment", the API name of Attachment
  // object.
  attachment.subjectType = 'Attachment';
  // Setting the fieldlist property with the
  // array of field names that should be saved
  // to Salesforce.
  attachment.fieldlist =
    ['ParentId', 'Name', 'Body'];
  // Setting the values of ParentId, Name and
  // Body fields on the object.
  attachment.set('ParentId',
    that.$.force_ui_detail.model.id);
  attachment.set('Name', 'camera.png');
  attachment.set('Body', blob);
  // Saving the attachment record to Salesforce.
  attachment.save();
}

// Instanting the FileReader JS API to read the
// contents of image file provided by the user.
var reader = new FileReader();
// Assigning the listener to the "loadend" event on
// FileReader. This event is thrown once the
// contents of the file are fully read into the
// memory.
reader.onloadend = function () {
  // Once the file contents are read, the
  // reader.result property returns the data url
  // of the content with the base64 encoded
  // contents. Using the reader.result property to
  // add that image to the view.
  $(that.$.attachments).append(
    '');
  // Now using the saveFileToSFDC method to save
  // the base64 string as attachment to
  // Salesforce.
  saveFileToSFDC(reader.result.split(',')[1]);
}
// Initiate the reading of file contents of the
// selected image. e.srcElement returns the input
// element on which the "change" event was fired.
reader.readAsDataURL(e.srcElement.files[0]);
}
});
</script>
</polymer-element>

```

After the mobile-app definition is complete, the app instantiates the Polymer object the app by simply adding a mobile-app node to the end of the body element.

```

<body>
  <!-- Define the Polymer element -->
  ...

```

```

    <!-- Add the newly defined "mobile-app" element to the
         page. This triggers polymer to enable the rendering
         of the "mobile-app" element based on the definition
         above. -->
    <mobile-app></mobile-app>
</body>

```

Doing this triggers Polymer to begin rendering the `mobile-app` element as HTML.

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 `forceck.js`

Using JavaScript Remoting to Invoke Your Apex Controller

Like `apex:actionFunction`, [JavaScript remoting](#) lets you invoke methods in your Apex controller through JavaScript code hosted on your Visualforce page.

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 [this Dreamforce 2012 session](#) for a more detailed comparison between the JavaScript remoting and `actionFunction`. See http://www.salesforce.com/us/developer/docs/apexcode/Content/apex_classes_annotation_RemoteAction.htm to read more about `@RemoteAction` annotations.

Accessing the Salesforce API with ForceTK and JQuery

When you call Salesforce REST APIs from Visualforce, you're calling to a different domain. This separation violates same-origin browser policy, which causes the browser to refuse the connection. The ForceTK JavaScript library works around same-origin policy restrictions by using the [AJAX Proxy](#) to give full access to the REST API. Since the AJAX proxy is present on all Visualforce hosts with an endpoint of the form <https://<abc>.na1.visual.force.com/services/proxy>, your Visualforce-hosted JavaScript can invoke it by passing the desired resource URL in an HTTP header.

To use the proxy service:

1. Send your request to <https://<domain>/services/proxy>, where *<domain>* is the domain of your current Visualforce page.
2. Use the following HTTP headers:

SalesforceProxy-Endpoint

URL of the request endpoint

SalesforceProxy-SID

Current user session ID

For tips on accessing this proxy through JavaScript, see [AJAX Proxy](#).

The following code sample uses the jQuery Mobile library for the user interface. To run this code, your Visualforce page must include jQuery and the ForceTK library. To add these resources:

1. Create an archive file, such as a ZIP file, that contains `app.js`, `forcetk.js`, `jquery.js`, and any other static resources your project requires.
2. In Salesforce, upload the archive file via **Your Name > App Setup > Develop > Static Resources**.

After obtaining an instance of the jQuery Mobile library, the sample code creates a ForceTK client object and initializes it with a session ID. It then calls the asynchronous ForceTK `query()` method to process a SOQL query. The query callback

function uses jQuery Mobile to display the first Name field returned by the query as HTML in an object with ID “accountname.” At the end of the Apex page, the HTML5 content defines the accountname element as a simple tag.

```
<apex:page>
<apex:includeScript value="{!URLFOR($Resource.static, 'jquery.js')}" />
<apex:includeScript value="{!URLFOR($Resource.static, 'forcetk.js')}" />
<script type="text/javascript">
    // Get a reference to jQuery that we can work with
    $j = jQuery.noConflict();

    // Get an instance of the REST API client and set the session ID
    var client = new forcetk.Client();
    client.setSessionToken('{!$Api.Session_ID}');

    client.query("SELECT Name FROM Account LIMIT 1", function(response) {
        $j('#accountname').html(response.records[0].Name);
    });
</script>
<p>The first account I see is <span id="accountname"></span></p>
</apex:page>
```



Note:

- Using the REST API—even from a Visualforce page—consumes API calls.
- SalesforceAPI 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 SmartSync Data Framework 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 SmartSync to Access Salesforce Objects](#) on page 200.

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 Force.com in offline situations.

- <http://blogs.developerforce.com/developer-relations/2011/06/using-html5-offline-with-forcecom.html>
- <http://blogs.developerforce.com/developer-relations/2013/03/using-javascript-with-force-com.html>

Introduction to Hybrid Development

Hybrid apps combine the ease of HTML5 Web app development with the power and features of the native platform. They run within the Salesforce Mobile Container, a native layer that translates the app into device-specific code.

Hybrid apps depend on HTML and JavaScript files. These files can be stored on the device or on the server.

- **Device**—Hybrid apps developed with the `forcetk.mobilesdk.js` library wrap a Web app inside the Salesforce Mobile Container. In this scenario, the JavaScript and HTML files are stored on the device.
- **Server** — Hybrid apps developed using Visualforce technology store their HTML and JavaScript files on the Salesforce server and are delivered through the Salesforce Mobile Container.

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>).

iOS Hybrid Development

In order to develop hybrid applications, you'll need to meet some of the prerequisites for both the iOS native and the vanilla HTML5 scenarios.

1. Make sure you meet the requirements for [HTML5 and Hybrid Development](#).
2. Follow the installation instructions for [iOS](#).
3. After installing Mobile SDK for iOS, create a new hybrid app as described in [Creating an iOS Project](#). For the `apptype` parameter:
 - Use `--apptype="hybrid_local"` for a hybrid app with all code in the local project. Put your HTML and JavaScript files in `${target.dir}/assets/www/`.
 - Use `--apptype="hybrid_remote"` for a hybrid app with code in a Visualforce app on the server

Android Hybrid Development

To develop hybrid applications, you'll need to meet some of the prerequisites for both the Android native and the plain HTML5 scenarios.

1. Make sure you meet the requirements for [HTML5 and Hybrid Development](#).
2. Follow the installation instructions for [Android Native](#).
3. After installing Mobile SDK for Android, create a new hybrid app as described in [Creating an Android Project](#). For the `apptype` parameter:
 - Use `--apptype="hybrid_local"` for a hybrid app with all code in the local project. Put your HTML and JavaScript files in `${target.dir}/assets/www/`.
 - Use `--apptype="hybrid_remote"` for a hybrid app with code in a Visualforce app on the server

JavaScript Files for Hybrid Applications

In Salesforce Mobile SDK 2.0, we've refactored some JavaScript files and added new ones to support SmartSync. JavaScript files reside in the `forcedotcom/SalesforceMobileSDK-Shared` repository on GitHub.

Refactored JavaScript Files

These files are now collected in the `cordova.force.js` file.

- `SFHybridApp.js`
- `SalesforceOAuthPlugin.js`
- `SmartStorePlugin.js`

New JavaScript Files

These files are new in Mobile SDK 2.0.

JavaScript File	Description
cordova.force.js	Contains plugins for hybrid apps using the Cordova libraries
SmartSync.js	The SmartSync Data Framework library

New External Dependencies

Mobile SDK 2.0 introduces new external dependencies.

External JavaScript File	Description
jquery.js	Popular HTML utility library
underscore.js	SmartSync support
backbone.js	SmartSync support

Which JavaScript Files Do I Include?

Files that you include depend on the type of hybrid project. For each type described here, include all files in the list.

For basic hybrid apps:

- cordova.js
- cordova.force.js

To make REST API calls from a basic hybrid app:

- cordova.js
- cordova.force.js
- forcetk.mobilesdk.js

To use SmartSync in a hybrid app:

- jquery.js
- underscore.js
- backbone.js
- cordova.js
- cordova.force.js
- forcetk.mobilesdk.js
- SmartSync.js

Hybrid Apps Quick Start

Use the following procedure to get started quickly.

1. To develop apps for Android, you need:

- Java JDK 6 or higher—<http://www.oracle.com/downloads>.
- Apache Ant 1.8 or later—<http://ant.apache.org>.
- Android SDK Tools, version 21 or later—<http://developer.android.com/sdk/installing.html>.



Note: For best results, install all previous versions of the Android SDK as well as your target version.

- Eclipse—<https://www.eclipse.org>. Check the [Android Development Tools website](#) for the minimum supported Eclipse version.
 - Android ADT (Android Development Tools) plugin for Eclipse, version 21 or later—<http://developer.android.com/sdk>.
 - In order to run the application in the Emulator, you need to set up at least one Android Virtual Device (AVD) that targets Platform 2.2 or above (we recommend 4.0 or above). To learn how to set up an AVD in Eclipse, follow the instructions at <http://developer.android.com/guide/developing/devices/managing-avds.html>.
2. To develop apps for iOS, you need:
 - Xcode—Version 5.0 is the minimum, but we recommend the latest version.
 - iOS 6.0 or higher.
 - Mac OS X 10.8 (“Mountain Lion”) or higher.
 - A Salesforce [Developer Edition organization](#) with a [connected app](#).
 3. Install the Mobile SDK.
 - [Android Installation](#) on page 21
 - [iOS Installation](#) on page 21
 4. [Create a Connected App](#) on page 18.



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 [Creating an Android Project](#) on page 141. Use `hybrid_remote` for the application type.
 - Follow the steps at [Creating an iOS Project](#) on page 102. Use `hybrid_remote` for the application type.
6. [Run the Sample App](#) on page 85.

When you’re done with the sample app you can add more functionality.

1. [Create a Mobile Page to List Information](#) on page 90
2. [Create a Mobile Page for Detailed Information](#) on page 93

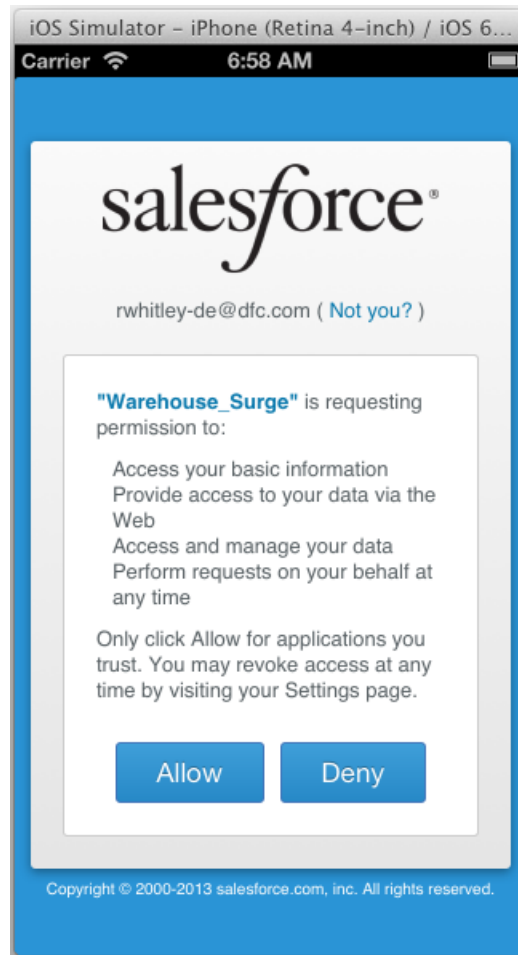
Running the Sample Hybrid App

You should now be able to compile and run the sample project, either on the simulator or a physical device. In both environments, you can select either a connected physical device or a simulator on which to run the app. If you’re using an iOS device, you must configure a profile as described in the Xcode User Guide at developer.apple.com/library. Similarly, Android devices must be set up developer.android.com/tools.

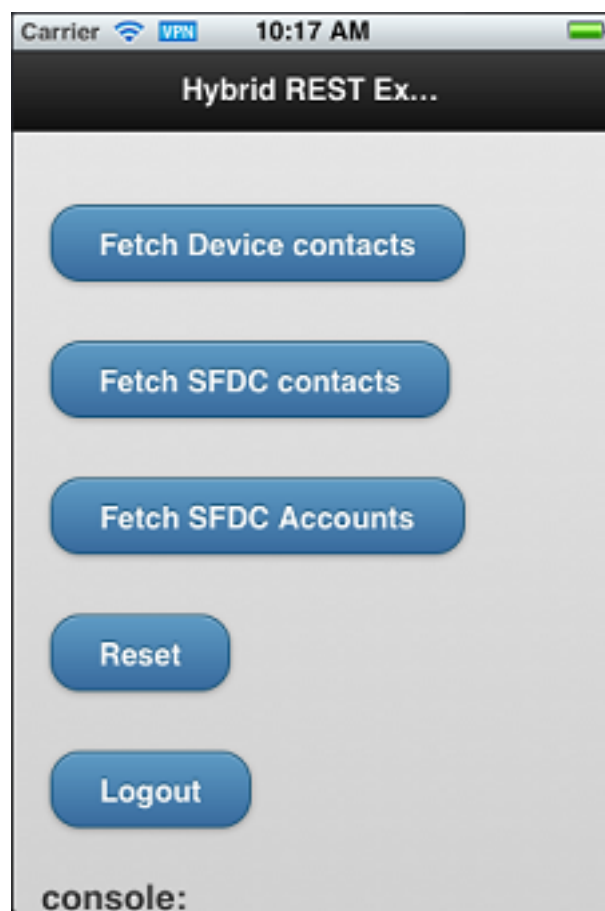
Whichever way you run the app, after showing an initial 'splash screen', you should see the Salesforce login screen.



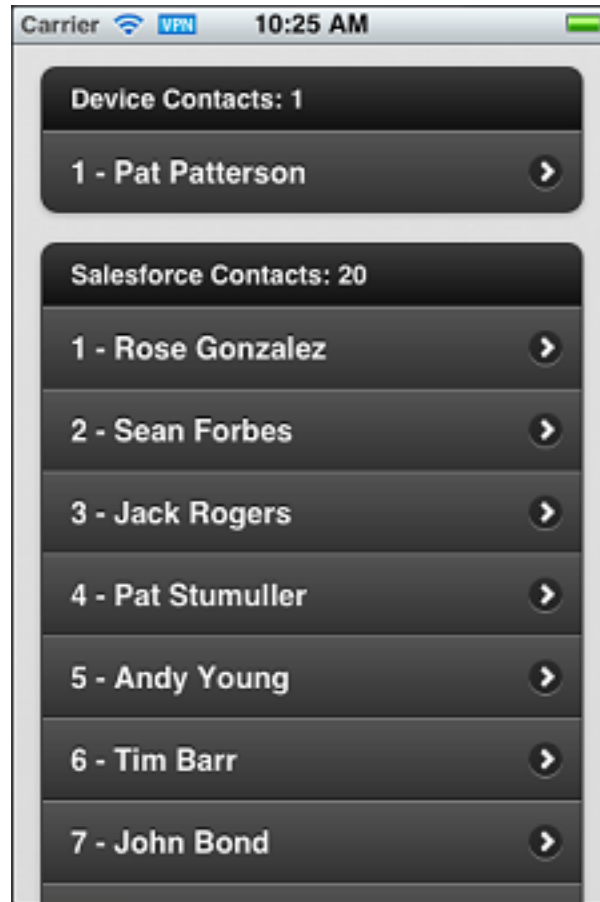
Log in with your DE username and password, and you will be prompted to allow your app access to your data in Salesforce.



Tap **Allow** and you should be able to retrieve lists of contacts and accounts from your DE account.



Tap to retrieve Contact and Account records from your DE account.



Notice the app can also retrieve contacts from the device - something that an equivalent web app would be unable to do. Let's take a closer look at how the app can do this.

How the Sample App Works

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 `onDeviceReady()` function calls `regLinkClickHandlers()` (in `inline.js`). `regLinkClickHandlers()` sets up five click handlers for the various functions in the sample app.

```
$j('#link_fetch_device_contacts').click(function() {
    SFHybridApp.logToConsole("link_fetch_device_contacts clicked");
    var options = new ContactFindOptions();
    options.filter = ""; // empty search string returns all contacts
    options.multiple = true;
    var fields = ["name"];
    navigator.contacts.find(fields, onSuccessDevice,
        onErrorDevice, options);
});
```

This handler calls `find()` on the `navigator.contacts` object to retrieve the contact list from the device. The `onSuccessDevice()` function renders the contact list into the `index.html` page.

```
$j('#link_fetch_sfcdc_contacts').click(function() {
    SFHybridApp.logToConsole("link_fetch_sfcdc_contacts clicked");
    forcetkClient.query("SELECT Name FROM Contact",
```

```
        onSuccessSfdcContacts, onErrorSfdc);
    });
```

The `#link_fetch_sfdc_contacts` handler runs a query using the `forcetkClient` object. This object is set up during the initial OAuth 2.0 interaction, and gives access to the Force.com REST API in the context of the authenticated user. Here we retrieve the names of all the contacts in the DE account and `onSuccessSfdcContacts()` renders them as a list on the `index.html` page.

```
$j('#link_fetch_sfdc_accounts').click(function() {
    SFHybridApp.logToConsole("link_fetch_sfdc_accounts clicked");
    forcetkClient.query("SELECT Name FROM Account",
        onSuccessSfdcAccounts, onErrorSfdc);
});
```

The `#link_fetch_sfdc_accounts` handler is very similar to the previous one, fetching Account records via the Force.com REST API. The remaining handlers, `#link_reset` and `#link_logout`, clear the displayed lists and log out the user respectively.

Create a Mobile Page to List Information

The sample hybrid app is useful in many respects, and serves as a good starting point to learn hybrid mobile app development. In this tutorial, you modify the sample hybrid mobile app to display Merchandise records in the custom Warehouse app schema.

You can build the Warehouse schema quickly using the getting started content online:

http://wiki.developerforce.com/page/Developing_Cloud_Apps_-_Coding_Optional. You can also install it as a package in your Developer Edition org: <https://login.salesforce.com/packaging/installPackage.apexp?p0=04ti00000000Pj8s>.

Modify the App's Initialization Block (index.html)

In this section, you modify the view file (`index.html`) and the controller (`inline.js`) to make the app specific to the Warehouse schema and display all records in the Merchandise custom object.

In your app, you want a list of Merchandise records to appear on the default Home page of the mobile app. Consequently, the first thing to do is to modify what happens automatically when the app calls the `onDeviceReady` function. Add the following code to the tail end of the sample `salesforceSessionRefreshed` function in `index.html`.

```
// log message
SFHybridApp.logToConsole("Calling out for records");
// register click event handlers -- see inline.js
regLinkClickHandlers();
// retrieve Merchandise records, including the Id for links
forcetkClient.query("SELECT Id, Name, Price__c, Quantity__c
    FROM Merchandise__c", onSuccessSfdcMerchandise, onErrorSfdc);
```

Notice that this JavaScript code leverages the ForceTK library to query the Force.com database with a basic SQL statement and retrieve all records from the Merchandise custom object. On success, the function calls the JavaScript function `onSuccessSfdcMerchandise` (which you build in a moment).

Create the App's mainpage View (index.html)

To display the Merchandise records in a standard mobile, touch-oriented user interface, scroll down in `index.html` and replace the entire `<body>` tag with the following HTML.

```
<!-- Main page, to display list of Merchandise once app starts -->
<div data-role="page" data-theme="b" id="mainpage">
  <!-- page header -->
  <div data-role="header">
    <!-- button for logging out -->
    <a href="#" id="link_logout" data-role="button"
      data-icon='delete'>
      Log Out
    </a>
    <!-- page title -->
    <h2>List</h2>
  </div>
  <!-- page content -->
  <div id="#content" data-role="content">
    <!-- page title -->
    <h2>Mobile Inventory</h2>
    <!-- list of merchandise, links to detail pages -->
    <div id="div_merchandise_list">
      <!-- built dynamically by function onSuccessSfdcMerchandise -->
    </div>
  </div>
</div>
```

Overall, notice that the updated view uses standard HTML tags and jQuery Mobile markup (e.g., `data-role`, `data-theme`, `data-icon`) to format an attractive touch interface for your app. Developing hybrid-based mobile apps is straightforward if you already know some basic standard Web development technology, such as HTML, CSS, JavaScript, and jQuery.

Modify the App's Controller (inline.js)

In the previous section, the initialization block in the view defers to the `onSuccessSfdcMerchandise` function of the controller to dynamically generate the HTML that renders Merchandise list items in the encompassing `div`, `div_merchandise_list`. In this step, you build the `onSuccessSfdcMerchandise` function.

Load the `inline.js` file and add the following controller action, which is somewhat similar to the sample functions.

```
// handle successful retrieval of Merchandise records
function onSuccessSfdcMerchandise(response) {
  // avoid jQuery conflicts
  var $j = jQuery.noConflict();

  // debug info to console
  SFHybridApp.logToConsole("onSuccessSfdcMerchandise: received " +
    response.totalSize + " merchandise records");

  // clear div_merchandise_list HTML
  $j("#div_merchandise_list").html("");

  // set the ul string var to a new UL
  var ul = $j('<ul data-role="listview" data-inset="true"
    data-theme="a" data-dividertheme="a"></ul>');

  // update div_merchandise_list with the UL
  $j("#div_merchandise_list").append(ul);

  // set the first li to display the number of records found
  // formatted using list-divider
  ul.append($j('<li data-role="list-divider">Merchandise records: '
    + response.totalSize + '</li>'));

  // add an li for the merchandise being passed into the function
```

```

// create array to store record information for click listener
inventory = new Array();
// loop through each record, using vars i and merchandise
$.each(response.records, function(i, merchandise) {
    // create an array element for each merchandise record
    inventory[merchandise.Id] = merchandise;
    // create a new li with the record's Name
    var newLi = $j("<li class='detailLink' data-id='" + merchandise.Id
        + "'><a href='#'>" + merchandise.Name + "</a></li>");
    ul.append(newLi);
});

// render (create) the list of Merchandise records
$j("#div_merchandise_list").trigger( "create" );
// send the rendered HTML to the log for debugging
SFHybridApp.logToConsole($j("#div_merchandise_list").html());

// set up listeners for detailLink clicks
$j(".detailLink").click(function() {
    // get the unique data-id of the record just clicked
    var id = $j(this).attr('data-id');
    // using the id, get the record from the array created above
    var record = inventory[id];

    // use this info to set up various detail page information
    $j("#name").html(record.Name);
    $j("#quantity").val(record.Quantity__c);
    $j("#price").val(record.Price__c);
    $j("#detailpage").attr("data-id", record.Id);

    // change the view to the detailpage
    $j.mobile.changePage('#detailpage', {changeHash: true});
});
}

```

The comments in the code explain each line. Notice the call to `SFHybridApp.logToConsole()`; the JavaScript outputs rendered HTML to the console log so that you can see what the code creates. Here's an excerpt of some sample output.

```

<ul data-role="listview" data-inset="true" data-theme="a"
    data-dividertHEME="a" class="ui-listview ui-listview-inset
    ui-corner-all ui-shadow">
<li data-role="list-divider" role="heading"
    class="ui-li ui-li-divider ui-btn ui-bar-a ui-corner-top">Merchandise records: 6
</li>
<li class="detailLink ui-btn ui-btn-up-a ui-btn-icon-right ui-li"
    data-id="a00E0000003BzSfIAK" data-theme="a">
<div class="ui-btn-inner ui-li">
<div class="ui-btn-text">
<a href="#" class="ui-link-inherit">Tablet</a>
</div>
</div>
</li>
<li class="detailLink ui-btn ui-btn-up-a ui-btn-icon-right ui-li"
    data-id="a00E0000003BuUpIAK" data-theme="a">
<div class="ui-btn-inner ui-li">
<div class="ui-btn-text">
<a href="#" class="ui-link-inherit">Laptop</a>
</div>
</div>
</li>
...
</ul>

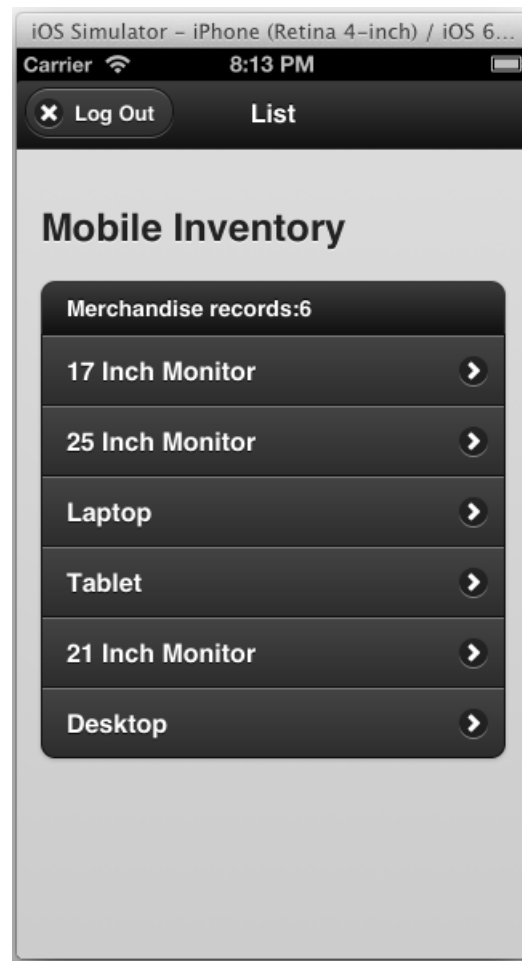
```

In particular, notice how the code:

- creates a list of Merchandise records for display on the app's primary page
- creates each list item to display the Name of the Merchandise record
- creates each list item with unique link information that determines what the target detail page displays

Test the New App

Restart the simulator for your mobile app. When you do, the initial page should look similar to the following screen.



If you click any particular Merchandise record, nothing happens yet. The list functionality is useful, but even better when paired with the detail view. The next section helps you build the *detailpage* that displays when a user clicks a specific Merchandise record.

Create a Mobile Page for Detailed Information

In the previous topic, you modified the sample hybrid app so that, after it starts, it lists all Merchandise records and provides links to detail pages. In this topic, you finish the job by creating a *detailpage* view and updating the app's controller.

Create the App's detailpage View (index.html)

When a user clicks on a Merchandise record in the app's *mainpage* view, click listeners are in place to generate record-specific information and then load a view named *detailpage* that displays this information. To create the *detailpage* view, add the following div tag after the *mainpage* div tag.

```
<!-- Detail page, to display details when user clicks specific Merchandise record -->

<div data-role="page" data-theme="b" id="detailpage">
  <!-- page header -->
  <div data-role="header">
    <!-- button for going back to mainpage -->
    <a href='#mainpage' id="backInventory"
      class='ui-btn-left' data-icon='home'>
      Home
    </a>
    <!-- page title -->
    <h1>Edit</h1>
  </div>
  <!-- page content -->
  <div id="#content" data-role="content">
    <h2 id="name"></h2>
    <label for="price" class="ui-hidden-accessible">
      Price ($):</label>
    <input type="text" id="price" readonly="readonly"></input>
    <br/>
    <label for="quantity" class="ui-hidden-accessible">
      Quantity:</label>
    <!-- note that number is not universally supported -->
    <input type="number" id="quantity"></input>
    <br/>
    <a href="#" data-role="button" id="updateButton"
      data-theme="b">Update</a>
  </div>
</div>
```

The comments explain each part of the HTML. Basically, the view is a form that lets the user see a Merchandise record's Price and Quantity fields, and optionally update the record's Quantity.

Recall, the jQuery calls in the last part of the `onSuccessSfdcMerchandise` function (in `inline.js`) and updates the detail page elements with values from the target Merchandise record. Review that code, if necessary.

Modify the App's Controller (inline.js)

What happens when a user clicks the Update button in the new *detailpage* view? Nothing, yet. You need to modify the app's controller (`inline.js`) to handle clicks on that button.

In `inline.js`, add the following JavaScript to the tail end of the `regLinkClickHandlers` function.

```
// handle clicks to Update on detailpage
$( "#updateButton" ).click(function() {
  // update local information in the inventory array
  inventory[$j("#detailpage").attr("data-id")].Quantity__c = $j("#quantity").val();
  currentRecord = inventory[$j("#detailpage").attr("data-id")];

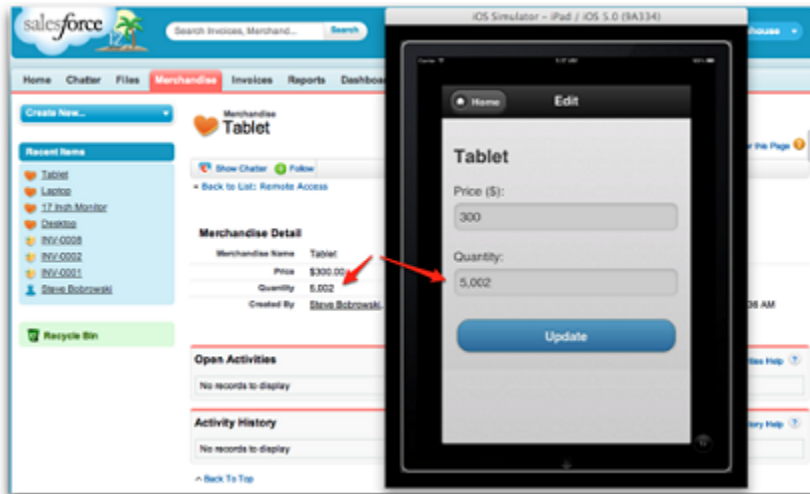
  // strip out ID before updating the database
  var data = new Object();
  data.Quantity__c = $j("#quantity").val();
  // update the database
  forcetkClient.update("Merchandise__c", currentRecord.Id,
    data,updateSuccess,onErrorSfdc);
});
```


The comments in the code explain each line. On success, the new handler calls the `updateSuccess` function, which is not currently in place. Add the following simple function to `inline.js`.

```
function updateSuccess(message) {
    alert("Item Updated");
}
```

Test the App

Restart the simulator for your mobile app. When you do, a detail page should appear when you click a specific Merchandise record and look similar to the following screen.



Feel free to update a record's quantity, and then check that you see the same quantity when you log into your DE org and view the record using the Force.com app UI (see above).

Guidelines and Tips for Hybrid Apps

When you create and deploy a hybrid Mobile SDK app, give special consideration to how you mix and distribute JavaScript libraries. Also, you can take steps in your code to reactively manage session expiration.

Versioning and Javascript Library Compatibility

In hybrid applications, client Javascript code interacts with native code through Cordova (formerly PhoneGap) and SalesforceSDK plugins. When you package your Javascript code with your mobile application, your testing assures that the code works with native code. However, when 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 PhoneGap or 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

Using the User Agent to Find the Mobile SDK Version

Fortunately, you can look up the Mobile SDK version in the user agent. 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 `sdkinf` Plugin

In Javascript, you can also retrieve the Mobile SDK version and other information by using the `sdkinf` plugin. This plugin, 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
<code>sdkVersion</code>	Version of the Salesforce Mobile SDK used to build to the container. For example: "1.4".
<code>appName</code>	Name of the hybrid application.
<code>appVersion</code>	Version of the hybrid application.
<code>forcePluginsAvailable</code>	Array containing the names of Salesforce plugins installed in the container. For example: "com.salesforce.oauth", "com.salesforce.smartstore", and so on.

The following code retrieves the information stored in the `sdkinf` plugin and displays it in alert boxes.

```
var sdkinf = cordova.require("salesforce/plugin/sdkinf");
sdkinf.getInfo(new function(info) {
```

```

alert("sdkVersion->" + info.sdkVersion);
alert("appName->" + info.appName);
alert("appVersion->" + info.appVersion);
alert("forcePluginsAvailable->" + JSON.stringify(info.forcePluginsAvailable));
});

```

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 additional 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, click **Develop** > **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 Applications

Mobile users expect their apps to just work. To help iron out common difficulties that plague many mobile apps, the Mobile SDK uses native containers for hybrid applications. These containers provide seamless authentication and session management by abstracting the complexity of web session management. 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.

Mobile SDK 1.4 begins to transition hybrid apps away from predefined, proactive session management to more flexible, reactive session management. Rather than letting the hybrid container automatically control the session, developers can participate in the management by responding to session events. This change gives developers more control over managing sessions in the Salesforce Touch Platform.

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	Proactive Behavior in SDK 1.3 and Earlier	Reactive Behavior in SDK 1.4	Steps for Upgrading Code
REST API	Background session refresh	Refresh from JavaScript	No change for <code>forcetk.mobilesdk.js</code> . For other frameworks, add refresh code.
JavaScript Remoting in Visualforce	Restart app	Refresh session and CSRF token from JavaScript	Catch timeout, then either reload page or load a new <code>iFrame</code> .
jQuery Mobile	Restart app	Reload page	Catch timeout, then reload page.

These sections provide detailed coding steps for each architecture.

REST APIs (Including Apex2REST)

If you’re writing or upgrading a hybrid app that leverages REST APIs, detect an expired session and request a new access token at the time the REST call is made. We encourage authors of apps based on this framework to leverage API wrapping libraries, such as `forcetk.mobilesdk.js`, to manage session retention.

The following code, from `index.html` in the `ContactExplorer` sample application, demonstrates the recommended technique. When the application first loads, call `getAuthCredentials()` on the Salesforce OAuth plugin, passing the handle to your refresh function (in this case, `salesforceSessionRefreshed`.) The OAuth plugin function calls your refresh function, passing it the session and refresh tokens. Use these returned values to initialize `forcetk.mobilesdk`.

- From the `onDeviceReady()` function:

```
cordova.require("salesforce/plugin/oauth").getAuthCredentials(salesforceSessionRefreshed,
getAuthCredentialsError);
```

- `salesforceSessionRefreshed()` function:

```
function salesforceSessionRefreshed(credsData) {
    forcetkClient = new forcetk.Client(credsData.clientId, credsData.loginUrl);
    forcetkClient.setSessionToken(credsData.accessToken, apiVersion,
credsData.instanceUrl);
    forcetkClient.setRefreshToken(credsData.refreshToken);
    forcetkClient.setUserAgentString(credsData.userAgent);
}
```

For the complete code, see the `ContactExplorer` sample application
(`SalesforceMobileSDK-Android\hybrid\SampleApps\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. You can 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 developers: Reloading the entire page might not provide the optimal user experience. If you want 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

In `hasSessionExpired()`, instead of fully reloading the page as follows:

```
window.location.reload();
```

Do something like this:

```
// Refresh oauth token
cordova.require("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
  <provider>.refresh(function() {
    <Retry call for a seamless user experience>;
  });

},
function(error) {
  console.log("Refresh failed");
}
);
```

jQuery Mobile

jQueryMobile makes Ajax calls to transfer data for rendering a page. If a session expires, a 302 error is masked by the framework. To recover, incorporate the following code to force a page refresh.

```
$(document).on('pageloadfailed', function(e, data) {
  console.log('page load failed');
  if (data.xhr.status == 0) {
    // reloading the VF page to initiate authentication
    window.location.reload();
  }
});
```

Chapter 4

Native iOS Development

In this chapter ...

- [iOS Native Quick Start](#)
- [Native iOS Requirements](#)
- [Creating an iOS Project](#)
- [Developing a Native iOS App](#)
- [Tutorial: Creating a Native iOS Warehouse App](#)
- [iOS Native Sample Applications](#)

Salesforce Mobile SDK delivers libraries and sample Xcode projects for developing mobile apps on iOS.

Two main things that the iOS native SDK provides are:

- Automation of the OAuth2 login process, making it easy to integrate OAuth with your app.
- Access to the REST API with infrastructure classes (including third-party libraries such as RestKit) to make that access as easy as possible.

When you create a native app using the forceios application, your project starts as a fully functioning native sample app. This simple app allows you to connect to a Salesforce organization and run a simple query. It doesn't do much, but it lets you know things are working as designed.

iOS Native Quick Start

Use the following procedure to get started quickly.

1. Make sure you meet all of the [native iOS requirements](#).
2. Install the [Mobile SDK for iOS](#). If you prefer, you can install the [Mobile SDK for iOS](#) from GitHub instead.
3. Run the [template app](#).

Native iOS Requirements

iOS development with Mobile SDK 2.2 requires the following software.

- Xcode—Version 5.0 is the minimum, but we recommend the latest version.
- iOS 6.0 or higher.
- Mac OS X 10.8 (“Mountain Lion”) or higher.

On the Salesforce side, you’ll also need:

- Salesforce Mobile SDK 2.2 for iOS. See [Install the Mobile SDK](#).
- A Salesforce [Developer Edition organization](#) with a [connected app](#).

Creating an iOS Project

To create a new app, use `forceios` again on the command line. You have two options for configuring your app.

- Configure your application options interactively as prompted by the `forceios` app.
- Specify your application options directly at the command line.

Specifying Application Options Interactively

To enter application options interactively, do one of the following:

- If you installed Mobile SDK globally, type `forceios create`.
- If you installed Mobile SDK locally, type `<forceios_path>/node_modules/.bin/forceios create`.

The `forceios` utility prompts you for each configuration value.

```
rwhitley-ltm1:Downloads rwhitley$ forceios create
Enter your application type (native, hybrid_remote, or hybrid_local): native
Enter your application name: MyNativeiOSApp
Enter your company identifier (com.mycompany): com.acme.goodapps
Enter your organization name (Acme, Inc.): GoodApps, Inc.
Enter the output directory for your app (defaults to the current directory):
Enter your Connected App ID (defaults to the sample app's ID):
Enter your Connected App Callback URI (defaults to the sample app's URI):
Creating app in /Users/rwhitley/Downloads/MyNativeiOSApp
Successfully created native app 'MyNativeiOSApp'.
```

Specifying Application Options Directly

You can also specify your configuration directly by typing the full `forceios` command string. To see usage information, type `forceios` without arguments. The list of available options displays:

```
$ forceios
Usage:
forceios create
```



```
--apptype=<Application Type> (native, hybrid_remote, hybrid_local)
--appname=<Application Name>
--companyid=<Company Identifier> (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> (Defaults to the current working directory)]
[--appid=<Salesforce App Identifier> (The Consumer Key for your app.
    Defaults to the sample app.)]
[--callbackuri=<Salesforce App Callback URL> (The Callback URL for your app.
    Defaults to the sample app.)]
```

Using this information, type `forceios create`, followed by your options and values. For example:

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

Running the New Project in XCode

Apps created with the `forceios` template are ready to run “right out of the box”. After the app creation script finishes, you can open and run the project in Xcode.

- 1. In Xcode, select **File > Open**.
- 2. Navigate to the output folder you specified.
- 3. Open your app’s `xcodeproj` file.
- 4. Click the **Run** button in the upper left corner to see your new app in action.

forceios Command Parameters

These are the descriptions of the `forceios` command parameters:

Parameter Name	Description
--apptype	One of the following: <ul style="list-style-type: none">• “native”• “hybrid_remote” (server-side hybrid app using VisualForce)• “hybrid_local” (client-side hybrid app that doesn’t use VisualForce)
--appname	Name of your application
--companyid	A unique identifier for your company. This value is concatenated with the app name to create a unique app identifier for publishing your app to the App Store. For example, “com.myCompany.apps”.
--organization	The formal name of your company. For example, “Acme Widgets, Inc.”.
--packagename	Package identifier for your application. For example, “com.acme.app”.
--startpage	(hybrid remote apps only) Server path to the remote start page. For example: <code>/apex/MyAppStartPage</code> .

Parameter Name	Description
<code>--outputdir</code>	(Optional) Folder in which you want your project to be created. If the folder doesn't exist, the script creates it. Defaults to the current working directory.
<code>--appid</code>	(Optional) Your connected app's Consumer Key. Defaults to the consumer key of the sample app.  Note: If you don't specify your own value here, you're required to change it in the app before you publish to the App Store.
<code>--callbackuri</code>	(Optional) Your connected app's Callback URL. Defaults to the callback URL of the sample app.  Note: <ul style="list-style-type: none"> If you don't specify your own value here, you're required to change it in the app before you publish to the App Store. If you accept the default value for <code>--appid</code>, be sure to also accept the default value for <code>--callbackuri</code>.
<code>--usesmartstore=true</code>	(Optional) Include only if you want to use SmartStore for offline data. Defaults to false if not specified.

Running the Xcode Project Template App

The Xcode project template includes a sample application you can run right away.

1. Press **Command-R** and the default template app runs in the iOS simulator.
2. On startup, the application starts the OAuth authentication flow, which results in an authentication page. Enter your credentials, and click **Login**.
3. Tap **Allow** when asked for permission

You should now be able to compile and run the sample project. It's a simple app that logs you into an org via OAuth2, issues a `select Name from Account` SOQL query, and displays the result in a `UITableView` instance.

Developing a Native iOS App

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

- Classes and interfaces that make it easy to call the Salesforce REST API
- Fully implemented OAuth login and passcode protocols
- SmartStore library for securely managing user data offline

The native iOS SDK requires you to be proficient in Objective-C coding. You also need to be familiar with iOS application development principles and frameworks. If you're a newbie, [Start Developing iOS Apps Today](#) is a good place to begin learning. See [Native iOS Requirements](#) for additional prerequisites.

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

About Login and Passcodes

To access Salesforce objects from a Mobile SDK app, the user logs into an organization on a Salesforce server. When the login flow begins, your app sends its 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. The Mobile SDK handles presentation of the login and passcode screens, as well as authentication handshakes. Your app doesn't have to do anything to display these screens. However, you do need to understand the login flow and how OAuth tokens are handled. See [About PIN Security](#) and [OAuth2 Authentication Flow](#).

About Memory Management

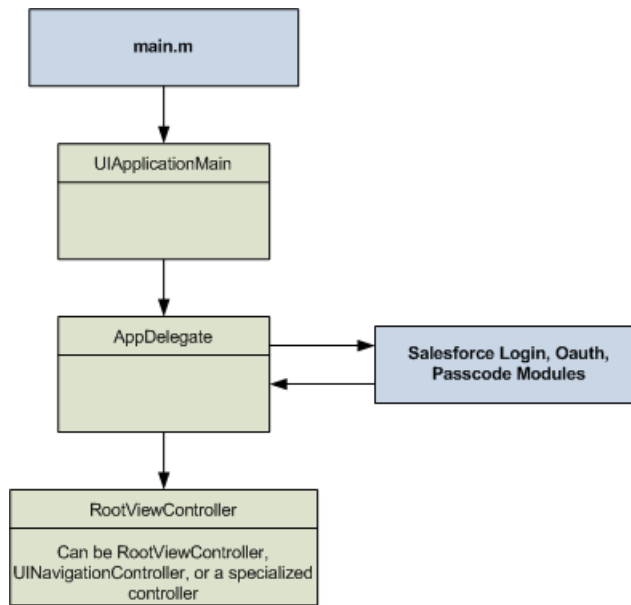
Beginning in Mobile SDK 2.0, native iOS apps use Automatic Reference Counting (ARC) to manage object memory. You don't have to allocate and then remember to deallocate your objects. See the [Mac Developer Library](#) at <https://developer.apple.com> for ARC syntax, guidelines, and best practices.

Overview of Application Flow

When you create a project with the forceios application, your new app defines three classes: `AppDelegate`, `InitialViewController`, and `RootViewController`. The `AppDelegate` object loads `InitialViewController` as the first view to show. After the authentication process completes, the `AppDelegate` object displays the view associated with `RootViewController` as the entry point to your app.

The workflow demonstrated by the template app is merely an example. You can tailor your `AppDelegate` and supporting classes to achieve your desired workflow. You can retrieve data through REST API calls and display it, launch other views, perform services, and so on. Your app remains alive in memory until the user quits it, or until the device is rebooted.

Native iOS apps built with the Mobile SDK follow the same design as other iOS apps. The `main.m` source file creates a `UIApplicationMain` object that is the root object for the rest of the application. The `UIApplicationMain` constructor creates an `AppDelegate` object that manages the application lifecycle.



AppDelegate Class

The `AppDelegate` class is the true entry point for an iOS app. In Mobile SDK apps, `AppDelegate` implements the standard iOS `UIApplicationDelegate` interface. The Mobile SDK template application for creating native iOS apps implements most of the Salesforce-specific startup functionality for you.

To customize the `AppDelegate` template, populate the following static variables with information from your Force.com Connected Application:

- `RemoteAccessConsumerKey`

```
static NSString * const RemoteAccessConsumerKey =
@"3MVG9Iu66FKeHhINkB1l7xt7kR8...YFDUpqRWcoQ2.dBv_a1Dyu5xa";
```

- `OAuthRedirectURI`

```
static NSString * const OAuthRedirectURI = @"testsfdc:///mobilesdk/detect/oauth/done";
```

OAuth functionality resides 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 `AppDelegate` implementation, or you can postpone login until it's actually required—for example, you can call `OAuth` from a sub-view.

Initialization

The following high-level overview shows how the `AppDelegate` initializes the template app. Keep in mind that you can change any of these details to suit your needs.

1. When the `[AppDelegate init]` message runs, it:

- Initializes configuration items, such as Connected App identifiers, OAuth scopes, and so on.
- Adds notification observers that listen to `SFAuthenticationManager`, `logoutInitiated`, and `loginHostChanged` notifications.

The `logoutInitiated` notification lets the app respond when a user logs out voluntarily or is logged out involuntarily due to invalid credentials. The `loginHostChanged` notification lets the app respond when the user changes the login

host (for example, from Production to Sandbox). See the `logoutInitiated:` and `loginHostChanged:` handler methods in the sample app.

- Initializes authentication "success" and "failure" blocks for the `[SFAuthenticationManager loginWithCompletion:failure:]` message. These blocks determine what happens when the authentication process completes.
- 2. `application:didFinishLaunchingWithOptions:`, a `UIApplicationDelegate` method, is called at app startup. The template app uses this method to initialize the `UIWindow` property, display the initial view (see `initializeAppViewState`), and initiate authentication. If authentication succeeds, the `SFAuthenticationManager` executes `initialLoginSuccessBlock` (the "success" block).
- 3. `initialLoginSuccessBlock` calls `setupRootViewController`, which creates and displays the app's `RootViewController`.

You can customize any part of this process. At a minimum, change `setupRootViewController` to display your own controller after authentication. You can also customize `initializeAppViewState` to display your own launch page, or the `InitialViewController` to suit your needs. You can also move the authentication details to where they make the most sense for your app. The Mobile SDK does not stipulate when—or if—actions must occur, but standard iOS conventions apply. For example, `self.window` must have a `rootViewController` by the time `application:didFinishLaunchingWithOptions:` completes.

UIApplication Event Handlers

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

`application:didFinishLaunchingWithOptions:`

First entry point when your app launches. Called only when the process first starts (not after a backgrounding/foregrounding cycle).

`applicationDidBecomeActive`

Called every time the application is foregrounded. The iOS SDK provides no default parent behavior; if you use it, you must implement it from the ground up.

For a list of all `UIApplication` event handlers, see "UIApplicationDelegate Protocol Reference" in the [iOS Developer Library](#).

About View Controllers

In addition to the views and view controllers discussed with the `AppDelegate` class, Mobile SDK exposes the `SFAuthorizingViewController` class. This controller displays the login screen when necessary.

To customize the login screen display:

1. Override the `SFAuthorizingViewController` class to implement your custom display logic.
2. Set the `[SFAuthenticationManager sharedManager].authViewController` property to an instance of your customized class.

The most important view controller in your app is the one that manages the first view that displays, after login or—if login is postponed—after launch. This controller is called your root view controller because it controls everything else that happens in your app. The Mobile SDK for iOS project template provides a skeletal class named `RootViewController` that demonstrates the minimal required implementation.

If your app needs additional view controllers, you're free to create them as you wish. The view controllers used in Mobile SDK projects reveal some possible options. For example, the Mobile SDK iOS template project bases its root view class on the

`UITableViewController` interface, while the `RestAPIExplorer` sample project uses the `UIViewController` interface. Your only technical limits are those imposed by iOS itself and the Objective-C language.

RootViewController Class

The `RootViewController` class exists only as part of the template project and projects generated from it. It implements the `SFRestDelegate` protocol to set up a framework for your app's interactions with the Salesforce REST API. Regardless of how you define your root view controller, it must implement `SFRestDelegate` if you intend to use it to access Salesforce data through the REST APIs.

RootViewController Design

As an element of a very basic app built with the Mobile SDK, 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

To do these things, the class inherits `UITableViewController` and implements the `SFRestDelegate` protocol. The action begins with an override of the `UIViewController:viewDidLoad` method:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.title = @"Mobile SDK Sample App";

    //Here we use a query that should work on either Force.com or Database.com
    SFRestRequest *request =
        [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name FROM User LIMIT 10"];
    [[SFRestAPI sharedInstance] send:request delegate:self];
}
```

The iOS runtime calls `viewDidLoad` only once in the view's life cycle, when the view is first loaded into memory. The intention in this skeletal app is to load only one set of data into the app's only defined view. If you plan to create other views, you might need to perform the query somewhere else. For example, if you add a detail view that lets the user edit data shown in the root view, you'll want to refresh the values shown in the root view when it reappears. In this case, you can perform the query in a more appropriate method, such as `viewWillAppear`.

After calling the superclass method, this code sets the title of the view, then issues a REST request in the form of an asynchronous `SOQL` query. The query in this case is a simple `SELECT` statement that gets the `Name` property from each `User` object and limits the number of rows returned to ten. Notice that the `requestForQuery` and `send:delegate:` messages are sent to a singleton shared instance of the `SFRestAPI` class. Use this singleton object for all REST requests. This object uses authenticated credentials from the singleton `SFAccountManager` object to form and send authenticated requests.

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

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

The `RootViewController` object can act as an `SFRestAPI` delegate because it implements the `SFRestDelegate` protocol. This protocol declares four possible response callbacks:

- `request:didLoadResponse:` — Your request was processed. The delegate receives the response in JSON format. This is the only callback that indicates success.
- `request:didFailLoadWithError:` — Your request couldn't be processed. The delegate receives an error message.
- `requestDidCancelLoad` — Your request was canceled by 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 one of the callbacks you've implemented in `RootViewController`. Place your code for handling Salesforce data in the `request:didLoadResponse:` callback. 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];
}
```

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.

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

- If you use invalid request parameters, you get a `kSFRestErrorDomain` error code. For example, if you pass `nil` to `requestForQuery:`, or you try 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, if 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 `RKRestKitErrorDomain` error code. For example, if a Salesforce server becomes temporarily inaccessible.

The other callbacks 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.

About Salesforce REST APIs

Native app development with the Salesforce Mobile SDK centers around the use of Salesforce REST APIs. Salesforce makes a wide range of object-based tasks available through URIs with REST parameters. Mobile SDK wraps these HTTP calls in interfaces that handle most of the low-level work in formatting a request.

In Mobile SDK for iOS, all REST requests are performed asynchronously. You can choose between delegate and block versions of the REST wrapper classes to adapt your requests to various scenarios. REST responses are formatted as `NSArray` or `NSDictionary` objects for a successful request, or `NSError` if the request fails.

See the [Force.com REST API Developer's Guide](#) for information on Salesforce REST response formats.

Supported Operations

The iOS REST APIs support the standard object operations offered by Salesforce REST and SOAP APIs. Salesforce Mobile SDK offers delegate and block versions of its REST request APIs. Delegate request methods are defined in the `SFRestAPI` class, while block request methods are defined in the `SFRestAPI (Blocks)` category. File requests are defined in the `SFRestAPI (Files)` category and are documented in [SFRestAPI \(Files\) Category](#).

Supported operations are:

Operation	Delegate method	Block method
Manual REST request Executes a request that you've built	send:delegate:	sendRESTRequest: failBlock: completeBlock:
SOQL query Executes the given SOQL string and returns the resulting data set	requestForQuery:	performSOQLQuery: failBlock: completeBlock:
SOSL search Executes the given SOSL string and returns the resulting data set	requestForSearch:	performSOSLSearch: failBlock: completeBlock:
Metadata Returns the object's metadata	requestForMetadataWithObjectType:	performMetadataWithObjectType: failBlock: completeBlock:
Describe global Returns a list of all available objects in your org and their metadata	requestForDescribeGlobal	performDescribeGlobalWithFailBlock: completeBlock:
Describe with object type Returns a description of a single object type	requestForDescribeWithObjectType:	performDescribeWithObjectType: failBlock: completeBlock:
Retrieve Retrieves a single record by object ID	requestForRetrieveWithObjectType: objectId: fieldList:	performRetrieveWithObjectType: objectId: fieldList: failBlock:completeBlock:
Update Updates an object with the given map	requestForUpdateWithObjectType: objectId: fields:	performUpdateWithObjectType: objectId: fields: failBlock: completeBlock:
Upsert Updates or inserts an object from external	requestForUpsertWithObjectType: externalIdField: externalId:	performUpsertWithObjectType: externalIdField: externalId: fields: failBlock: completeBlock:

Operation	Delegate method	Block method
data, based on whether the external ID currently exists in the external ID field		fields:
Create Creates a new record in the specified object	requestForCreateWithObjectType: fields:	performCreateWithObjectType: fields: failBlock: completeBlock:
Delete Deletes the object of the given type with the given ID	requestForDeleteWithObjectType: objectId:	performDeleteWithObjectType: objectId: failBlock: completeBlock:
Versions Returns Salesforce version metadata	requestForVersions	performRequestForVersionsWithFailBlock: completeBlock:
Resources Returns available resources for the specified API version, including resource name and URI	requestForResources	performRequestForResourcesWithFailBlock: completeBlock:

SFRestAPI Interface

SFRestAPI defines the native interface for creating and formatting Salesforce REST requests. It works by formatting and sending your requests to the Salesforce service, then relaying asynchronous responses to your implementation of the SFRestDelegate protocol.

SFRestAPI serves as a factory for SFRestRequest instances. It defines a group of methods that represent the request types supported by the Salesforce REST API. Each SFRestAPI method corresponds to a single request type. Each of these methods returns your request in the form of an SFRestRequest instance. You then use that return value to send your request to the Salesforce server. The HTTP coding layer is encapsulated, so you don't have to worry about REST API syntax.

For a list of supported query factory methods, see [Supported Operations](#)

SFRestDelegate Protocol

When a class adopts the SFRestDelegate protocol, it intends to be a target for REST responses sent from the Salesforce server. When you send a REST request to the server, you tell the shared SFRestAPI instance which object receives the response. When the server sends the response, Mobile SDK routes the response to the appropriate protocol method on the given object.

The SFRestDelegate protocol declares four possible responses:

- `request:didLoadResponse:` — Your request was processed. The delegate receives the response in JSON format. This is the only callback that indicates success.
- `request:didFailLoadWithError:` — Your request couldn't be processed. The delegate receives an error message.
- `requestDidCancelLoad` — Your request was canceled by 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 your implementation of one of these delegate methods. Because you don't know which type of response to expect, you must implement all of the methods.

request:didLoadResponse: Method

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];
}
```

At the server, all responses originate as JSON strings. Mobile SDK receives these raw responses and reformats them as iOS SDK objects before passing them to the `request:didLoadResponse:` method. Thus, the `jsonResponse` payload arrives as either an `NSDictionary` object or an `NSArray` object. The object type depends on the type of JSON data returned. If the top level of the server response represents a JSON object, `jsonResponse` is an `NSDictionary` object. If the top level represents a JSON array of other data, `jsonResponse` is an `NSArray` object.

If your method cannot infer the data type from the request, use `[NSObject isKindOfClass:]` 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".

request:didFailLoadWithError: Method

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, you pass `nil` to `requestForQuery:`, or you try 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 `RKRestKitErrorDomain` error code. For example, a Salesforce server becomes temporarily inaccessible.

requestDidCancelLoad and requestDidTimeout Methods

The `requestDidCancelLoad` and `requestDidTimeout` 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.

Creating REST Requests

Salesforce Mobile SDK for iOS natively supports many types of SOQL and SOSL REST requests. The `SFRestAPI` class provides factory methods that handle most of the syntactical details for you. Mobile SDK also offers considerable flexibility for how you create REST requests.

- For standard SOQL queries and SOSL searches, `SFRestAPI` methods create query strings based on minimal data input and package them in an `SFRestRequest` object that can be sent to the Salesforce server.
- If you are using a Salesforce REST API that isn't based on SOQL or SOSL, `SFRestRequest` methods let you configure the request itself to match the API format.
- The `SFRestAPI (QueryBuilder)` category provides methods that create free-form SOQL queries and SOSL search strings so you don't have to manually format the query or search string.
- Request methods in the `SFRestAPI (Blocks)` category let you pass callback code as block methods, instead of using a delegate object.

Sending a REST Request

Salesforce Mobile SDK for iOS natively supports many types of SOQL and SOSL REST requests. Luckily, the `SFRestAPI` provides factory methods that handle most of the syntactical details for you.

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

To send a REST request to the Salesforce server from an `SFRestAPI` delegate:

1. Build a SOQL, SOSL, or other REST request string.

For standard SOQL and SOSL queries, it's most convenient and reliable to use the factory methods in the `SFRestAPI` class. See [Supported Operations](#).

2. Create an `SFRestRequest` object with your request string.

Message the `SFRestAPI` singleton with the request factory method that suits your needs. For example, this code uses the `theSFRestAPI:requestForQuery:` method, which prepares a SOQL query.

```
// Send a request factory message to the singleton SFRestAPI instance
SFRestRequest *request = [[SFRestAPI sharedInstance]
    requestForQuery:@"SELECT Name FROM User LIMIT 10"];
```

3. Send the `send:delegate:` message to the shared `SFRestAPI` instance. Use your new `SFRestRequest` object as the `send:` parameter. The second parameter designates an `SFRestDelegate` object to receive the server's response. In the following example, the class itself implements the `SFRestDelegate` protocol, so it sets `delegate:` to `self`.

```
// Use the singleton SFRestAPI instance to send the
// request, specifying this class as the delegate.
[[SFRestAPI sharedInstance] send:request delegate:self];
```

SFRestRequest Class

Salesforce Mobile SDK provides the `SFRestRequest` interface as a convenience class for apps. `SFRestAPI` provides request methods that use your input to form a request. This request is packaged as an `SFRestRequest` instance and returned to your app. In most cases you don't manipulate the `SFRestRequest` object. Typically, you simply pass it unchanged to the `SFRestAPI:send:delegate:` method.

If you're sending a REST request that isn't directly supported by the Mobile SDK—for example, if you want to use the Chatter REST API—you can manually create and configure an `SFRestRequest` object.

Using SFRestRequest Methods

`SFRestAPI` tools support SOQL and SOSL statements natively: they understand the grammar and can format valid requests based on minimal input from your app. However, Salesforce provides some product-specific REST APIs that have no relationship to SOQL queries or SOSL searches. You can still use Mobile SDK resources to configure and send these requests. This process is similar to sending a SOQL query request. The main difference is that you create and populate your `SFRestRequest` object directly, instead of relying on `SFRestAPI` methods.

To send a non-SOQL and non-SOSL REST request using the Mobile SDK:

1. Create an instance of `SFRestRequest`.
2. Set the properties you need on the `SFRestRequest` object.
3. Call `send:delegate:` on the singleton `SFRestAPI` instance, passing in the `SFRestRequest` object you created as the first parameter.

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

```
SFRestRequest *request = [[SFRestRequest alloc] init];
[request setDelegate:self];
[request setEndpoint:kSFDefaultRestEndpoint];
[request setMethod:SFRestMethodGET];
[request setPath:
    [NSString stringWithFormat:@"%v26.0/chatter/feeds/record/%@/feed-items",
    recordId]];
[[SFRestAPI sharedInstance] send:request delegate:self];
```

4. Alternatively, you can create the same request using the `requestWithMethod:path:queryParams` class method.

```
SFRestRequest *request =
    [SFRestRequest
        requestWithMethod:SFRestMethodGET
        path:
            [NSString
                stringWithFormat:
                    @"%v26.0/chatter/feeds/
                    record/%@/feed-items",
                    recordId]
        queryParams:nil];
[[SFRestAPI sharedInstance] send:request delegate:self];
```

5. To perform a request with parameters, create a parameter string, and then use the `SFJsonUtils:objectFromJSONString` static method to wrap it in an `NSDictionary` object. (If you prefer, you can create your `NSDictionary` object directly, before the method call, instead of creating it inline.)

The following example performs a POST operation that adds a comment to a Chatter feed.

```
NSString *body =
    [NSString stringWithFormat:
        @"{ \"body\" :
            {\"messageSegments\" :
                [{ \"type\" : \"Text\",
                    \"text\" : \"%@\"}]
            },
        },
        comment];

SFRestRequest *request =
    [SFRestRequest
        requestWithMethod:SFRestMethodPOST
        path:[NSString
            stringWithFormat:
                @\"/v26.0/chatter/feeds/
                record/%@/feed-items\",
                recordId]
        queryParams:
            (NSDictionary *)
            [SFJsonUtils objectFromJSONString:body]];
[[SFRestAPI sharedInstance] send:request delegate:self];
```

6. 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. If you find that your native app displays unwanted escape sequences in Chatter comments, set the `X-Chatter-Entity-Encoding` header to "false" before sending your request, as follows:

```
...
[request setHeaderValue:@"false" forHeaderName:@"X-Chatter-Entity-Encoding"];
[[SFRestAPI sharedInstance] send:request delegate:self];
```

SFRestAPI (Blocks) Category

If you prefer, you can use blocks instead of a delegate to execute callback code. Salesforce Mobile SDK for native iOS provides a block corollary for each `SFRestAPI` request method. These methods are defined in the `SFRestAPI (Blocks)` category.

Block request methods look a lot like delegate request methods. They all return a pointer to `SFRestRequest`, and they require the same parameters. Block request methods differ from their delegate siblings in these ways:

1. In addition to copying the REST API parameters, each method requires two blocks: a fail block of type `SFRestFailBlock`, and a complete block of type `SFRestDictionaryResponseBlock` or type `SFRestArrayResponseBlock`, depending on the expected response data.
2. Block-based methods send your request for you, so you don't need to call a separate send method. If your request fails, you can use the `SFRestRequest * return value` to retry the request. To do this, use the `SFRestAPI:sendRESTRequest:failBlock:completeBlock:` method.

Judicious use of blocks and delegates can help fine-tune your app's readability and ease of maintenance. Prime conditions for using blocks often correspond to those that mandate inline functions in C++ or anonymous functions in Java. However, this observation is just a general suggestion. Ultimately, you need to make a judgement call based on research into your app's real-world behavior.

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 value to one of the following `SFRestAPI` methods.

- `-(SFRestRequest *)requestForQuery:(NSString *)soql;`
- `-(SFRestRequest *)requestForSearch:(NSString *)sosl;`

`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.

SOSL Methods

SOSL query builder methods are:

```
+ (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:

- `term` is the search string. This string can be any arbitrary value. The method escapes any SOSL reserved characters before processing the search.
- `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 \(SOSL\)](#).
- `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.

- `limit`—If you want 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:

```
+ (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 `fields` and `sObject` can be set to `nil`.

Parameter name	Description
<code>fields</code>	An array of field names to be queried.
<code>sObject</code>	Name of the object to query.
<code>where</code>	An expression specifying one or more query conditions.
<code>groupBy</code>	An array of field names to use for grouping the resulting records.
<code>having</code>	An expression, usually using an aggregate function, for filtering the grouped results. Used only with <code>groupBy</code> .
<code>orderBy</code>	An array of fields name to use for ordering the resulting records.
<code>limit</code>	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:

```
+ (NSString *) sanitizeSOSLSearchTerm:(NSString *)searchTerm;
```

This method escapes every SOSL reserved character in the input string, and returns the escaped version. For example:

```
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 create file operation requests. Each method returns a new `SFRestRequest` object. Applications send this object to the Salesforce service to process the request. 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];
```



Note: 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.

See [Files and Networking](#) for a full description of the Files feature and networking functionality.

Methods

`SFRestAPI (Files)` category supports the following operations. For a full reference of this category, see `SFRestAPI (Files) Category—Request Methods (iOS)`. For a full description of the REST request and response bodies, go to **Chatter REST API Resources > FilesResources** at <http://www.salesforce.com/us/developer/docs/chatterapi>.

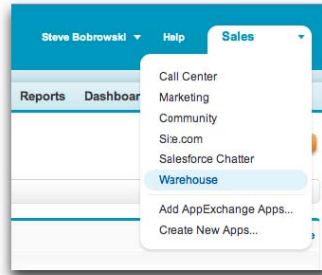
Method	Description
<pre>- (SFRestRequest *) requestForOwnedFilesList: (NSString *) userId page: (NSUInteger)page</pre>	Builds a request that fetches a page from the list of files owned by the specified user.
<pre>- (SFRestRequest *) requestForFilesInUsersGroups: (NSString *)userId page: (NSUInteger)page</pre>	Builds a request that fetches a page from the list of files owned by the user's groups.
<pre>- (SFRestRequest *) requestForFilesSharedWithUser: (NSString *)userId page: (NSUInteger)page</pre>	Builds a request that fetches a page from the list of files that have been shared with the user.
<pre>- (SFRestRequest *) requestForFileDetails: (NSString *)sfdcId forVersion: (NSString *)version</pre>	Builds a request that fetches the file details of a particular version of a file.
<pre>- (SFRestRequest *) requestForBatchFileDetails: (NSArray *)sfdcIds</pre>	Builds a request that fetches the latest file details of one or more files in a single request.
<pre>- (SFRestRequest *) requestForFileRendition: (NSString *)sfdcId</pre>	Builds a request that fetches the a preview/rendition of a particular page of the file (and version).

Method	Description
<pre>version: (NSString *)version renditionType: (NSString *)renditionType page: (NSUInteger)page</pre>	
<pre>- (SFRestRequest *) requestForFileContents: (NSString *) sfdcId version: (NSString*) version</pre>	Builds a request that fetches the actual binary file contents of this particular file.
<pre>- (SFRestRequest *) requestForFileShares: (NSString *)sfdcId page: (NSUInteger)page</pre>	Builds a request that fetches a page from the list of entities that share this file.
<pre>- (SFRestRequest *) requestForAddFileShare: (NSString *) fileId entityId: (NSString *)entityId shareType: (NSString*)shareType</pre>	Builds a request that add a file share for the specified file ID to the specified entity ID.
<pre>- (SFRestRequest *) requestForDeleteFileShare: (NSString *)shareId;</pre>	Builds a request that deletes the specified file share.
<pre>- (SFRestRequest *) requestForUploadFile: (NSData *)data name: (NSString *)name description: (NSString *)description mimeType: (NSString *)mimeType</pre>	Builds a request that uploads a new file to the server. Creates a new file with version set to 1.

Tutorial: Creating a Native iOS Warehouse App

Prerequisites

- This tutorial uses a Warehouse app that contains a basic inventory database. You'll need to install this app in a DE org. If you install it in an existing DE org, be sure to delete any existing Warehouse components you've made before you install.
 - Click the installation URL link: <https://login.salesforce.com/package/installPackage.apexp?p0=04ti0000000MMMT>
 - If you aren't logged in already, enter the username and password of your DE org.
 - On the Package Installation Details page, click **Continue**.
 - Click **Next**, and on the Security Level page click **Next**.
 - Click **Install**.
 - Click **Deploy Now** and then **Deploy**.
 - Once the installation completes, you can select the **Warehouse** app from the app picker in the upper right corner.



8. To create data, click the **Data** tab.
9. Click the **Create Data** button.

- Install the latest versions of Xcode and the iOS SDK.
- Install the Salesforce Mobile SDK using npm:
 1. If you've already successfully installed Node.js and npm, skip to step 4.
 2. Install Node.js on your system. The Node.js installer automatically installs npm.
 - i. Download Node.js from www.nodejs.org/download.
 - ii. Run the downloaded installer to install Node.js and npm. Accept all prompts asking for permission to install.
 3. At the Terminal window, type `npm` and press `Return` to make sure your installation was successful. If you don't see a page of usage information, revisit Step 2 to find out what's missing.
 4. At the Terminal window, type `sudo npm install forceios -g`

This command uses the `forceios` package to install the Mobile SDK globally. With the `-g` option, you can run `npm install` from any directory. The `npm` utility installs the package under `/usr/local/lib/node_modules`, and links binary modules in `/usr/local/bin`. Most users need the `sudo` option because they lack read-write permissions in `/usr/local`.

Create a Native iOS App

In this tutorial, you learn how to get started with the Salesforce Mobile SDK, including how to install the SDK and a quick tour of the native project template using your DE org. Subsequent tutorials show you how to modify the template app and make it work with the Warehouse schema.

Step 1: Create a Connected App

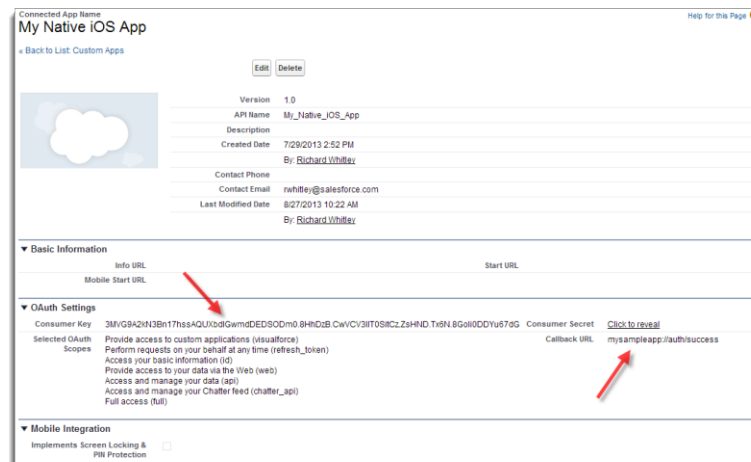
In this step, you learn how to configure a Connected App in Force.com. Doing so authorizes the mobile app you will soon build to communicate securely with Force.com and access Force.com APIs on behalf of users via the industry-standard OAuth 2.0 protocol.

1. In your DE org, click **Your Name** > **Setup** and under App Setup, click **Create** > **Apps**.
2. Under **Connected Apps**, click **New** to bring up the **New Connected App** page.
3. Under **Basic Information**, fill out the form as follows:
 - **Connected App Name:** My Native iOS App
 - **API Name:** accept the suggested value
 - **Contact Email:** enter your email address
4. Under OAuth Settings, check the **Enable OAuth Settings** checkbox.

5. Set **Callback URL** to `mysampleapp://auth/success`.
6. Under **Available OAuth Scopes**, check “Access and manage your data (api)” and “Perform requests on your behalf at any time (refresh_token)”, then click **Add**.
7. Click **Save**.

After you save the configuration, notice the details of the Connected App you just created.

- Note the Callback URL and Consumer Key; you will use these when you set up your native app in the next step.
- Mobile apps do not use the Consumer Secret, so you can ignore this value.



Step 2: Create a Native iOS Project

To create a new Mobile SDK project, use the `forceios` utility again in the Terminal window.

1. Change to the directory in which you want to create your project.
2. To create an iOS project, type `forceios create`.

The `forceios` utility prompts you for each configuration value.

3. For application type, enter `native`.
4. For application name, enter `MyNativeiOSApp`.
5. For company identifier, enter `com.acme.goodapps`.
6. For organization name, enter `GoodApps, Inc.`.
7. For output directory, enter `tutorial/iOSNative`.
8. For the Connected App ID, copy and paste the Consumer Key from your Connected App definition.
9. For the Connected App Callback URI, copy and paste the Callback URL from your Connected App definition.

The input screen should look similar to this:

```
rwhitley-ltm1:~ rwhitley$ forceios create
Enter your application type (native, hybrid_remote, or hybrid_local): native
Enter your application name: MyNativeiOSApp
Enter your company identifier (com.mycompany): com.acme.goodapps
Enter your organization name (Acme, Inc.): GoodApps, Inc.
Enter the output directory for your app (defaults to the current directory): tutorial/iOSNative
Enter your Connected App ID (defaults to the sample app's ID): 3MVG9A2kN3Bn17hssAQUXbdLGwmdDEDS0Dm0.8HhDzB.
CwVCV3lIT0SItCz.ZsHND.Tx6N.8Goli0DDYU67dG
Enter your Connected App Callback URI (defaults to the sample app's URI): https://login.salesforce.com/services/oauth2/success
Creating output folder tutorial/iOSNative
Creating app in /Users/rwhitley/tutorial/iOSNative/MyNativeiOSApp
Successfully created native app 'MyNativeiOSApp'.
```

Step 3: Run the New iOS App

1. In Xcode, select **File > Open**.
2. Navigate to the output folder you specified.
3. Open your app's `xcodeproj` file.
4. Click the **Run** button in the upper left corner to see your new app in the iOS simulator. Make sure that you've selected **Product > Destination > iPhone 6.0 Simulator** in the Xcode menu.
5. When you start the app, after showing an initial splash screen, you should see the Salesforce login screen. Login with your DE username and password.



6. When prompted, click **Allow** to let the app access your data in Salesforce. You should see a table listing the names of users defined in your DE org.



Step 4: Explore How the iOS App Works

The native iOS app uses a straightforward Model View Controller (MVC) architecture.

- The model is the Force.com database schema
- The views come from the nib and implementation files in your project
- The controller functionality represents a joint effort between the iOS SDK classes, the Salesforce Mobile SDK, and your app

AppDelegate Class and the Root View Controller

When the app is launched, the `AppDelegate` class initially controls the execution flow. After the login process completes, the `AppDelegate` instance passes control to the root view. In the template app, the root view controller class is named `RootViewController`. This class becomes the root view for the app in the `AppDelegate.m` file, where it's subsumed by a `UINavigationController` instance that controls navigation between views:

```
- (void)setupRootViewController
{
    RootViewController *rootVC = [[RootViewController alloc] initWithNibName:nil bundle:nil];

    UINavigationController *navVC = [[UINavigationController alloc]
initWithRootViewController:rootVC];
    self.window.rootViewController = navVC;
}
```

Before it's customized, though, the app doesn't include other views or touch event handlers. It simply logs into Salesforce, issues a request using Salesforce Mobile SDK REST APIs, and displays the response in the root view.

UITableViewController Class

RootViewController inherits the UITableViewController class. Because it doesn't customize the table in its inherited view, there's no need for a nib or xib file. The controller class simply loads data into the tableView property and lets the super class handle most of the display tasks. However, RootViewController does add some basic cell formatting by calling the tableView:cellForRowAtIndexPath: method. It creates a new cell, assigns it a generic ID (@"CellIdentifier"), puts an icon at the left side of the cell, and adds an arrow to the right side. Most importantly, it sets the cell's label to assume the Name value of the current row from the REST response object. Here's the code:

```
// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"CellIdentifier";

    // Dequeue or create a cell of the appropriate type.
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleValue1 reuseIdentifier:CellIdentifier] autorelease];
    }

    //if you want to add an image to your cell, here's how
    UIImage *image = [UIImage imageNamed:@"icon.png"];
    cell.imageView.image = image;

    // Configure the cell to show the data.
    NSDictionary *obj = [self.dataRows objectAtIndex:indexPath.row];
    cell.textLabel.text = [obj objectForKey:@"Name"];

    //this adds the arrow to the right hand side.
    cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

    return cell;
}
```

SFRestAPI Shared Object and SFRestRequest Class

You can learn how the app creates and sends the REST request by browsing the RootViewController.viewDidLoad method. The app defines a literal SOQL query string and passes it to the SFRestAPI:requestForQuery: instance method. To call this method, the app sends a message to the shared singleton SFRestAPI instance. The method creates and returns an appropriate, pre-formatted SFRestRequest object that wraps the SOQL query. The app then forwards this object to the server by sending the send:delegate: message to the shared SFRestAPI object:

```
//Here we use a query that should work on either Force.com or Database.com
SFRestRequest *request = [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name FROM User LIMIT 10"];
[[SFRestAPI sharedInstance] send:request delegate:self];
```

The SFRestAPI class serves as a factory for SFRestRequest instances. It defines a series of request methods that you can call to easily create request objects. If you want, you can also build SFRestRequest instances directly, but, for most cases, manual construction isn't necessary.

Notice that the app specifies self for the delegate argument. This tells the server to send the response to a delegate method implemented in the RootViewController class.

SFRestDelegate Interface

To be able to accept REST responses, `RootViewController` implements the `SFRestDelegate` interface. This interface declares four methods—one for each possible response type. The `request:didLoadResponse:` delegate method executes when the request succeeds. When `RootViewController` receives a `request:didLoadResponse:` callback, it copies the returned records into its data rows and reloads the data displayed in the Warehouse view. Here's the code that implements the `SFRestDelegate` interface in the `RootViewController` class:

```
#pragma mark - SFRestAPIDelegate

- (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];
}

- (void)request:(SFRestRequest*)request didFailLoadWithError:(NSError*)error {
    NSLog(@"request:didFailLoadWithError: %@", error);
    //add your failed error handling here
}

- (void)requestDidCancelLoad:(SFRestRequest *)request {
    NSLog(@"requestDidCancelLoad: %@", request);
    //add your failed error handling here
}

- (void)requestDidTimeout:(SFRestRequest *)request {
    NSLog(@"requestDidTimeout: %@", request);
    //add your failed error handling here
}
```

As the comments indicate, this code fully implements only the `request:didLoadResponse:` success delegate method. For responses other than success, this template app simply logs a message.

Customize the List Screen

In this tutorial, you modify the root view controller to make the app specific to the Warehouse schema. You also adapt the existing SQL query to obtain all the information we need from the Merchandise custom object.

Step 1: Modify the Root View Controller

To adapt the template project to our Warehouse design, let's rename the `RootViewController` class.

1. In the Project Navigator, choose the `RootViewController.h` file.
2. In the Editor, click the name “`RootViewController`” on this line:

```
@interface RootViewController : UITableViewController <SFRestDelegate>{
```

3. Using the Control-Click menu, choose **Refactor > Rename**. Be sure that **Rename Related Files** is checked.
4. Change “`RootViewController`” to “`WarehouseViewController`”. Click **Preview**.

Xcode presents a new window that lists all project files that contain the name “`RootViewController`” on the left. The central pane shows a diff between the existing version and the proposed new version of each changed file.

5. Click **Save**.
6. Click **Enable** when Xcode asks you if you'd like it to take automatic snapshots before refactoring.

After the snapshot is complete, the Refactoring window goes away, and you're back in your refactored project. Notice that the file names `RootViewController.h` and `RootViewController.m` are now `WarehouseViewController.h` and `WarehouseViewController.m`. Every instance of `RootViewController` in your project code has also been changed to `WarehouseViewController`.

Step 2: Create the App's Root View

The native iOS template app creates a SOQL query that extracts Name fields from the standard User object. For this tutorial, though, you use records from a custom object. Later, you create a detail screen that displays Name, Quantity, and Price fields. You also need the record ID.

Let's update the SOQL query to operate on the custom `Merchandise__c` object and to retrieve the fields needed by the detail screen.

1. In the Project Navigator, select `WarehouseViewController.m`.
2. Scroll to the `viewDidLoad` method.
3. Update the view's display name to "Warehouse App". Change:

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

to

```
self.title = @"Warehouse App"
```

4. Change the SOQL query in the following line:

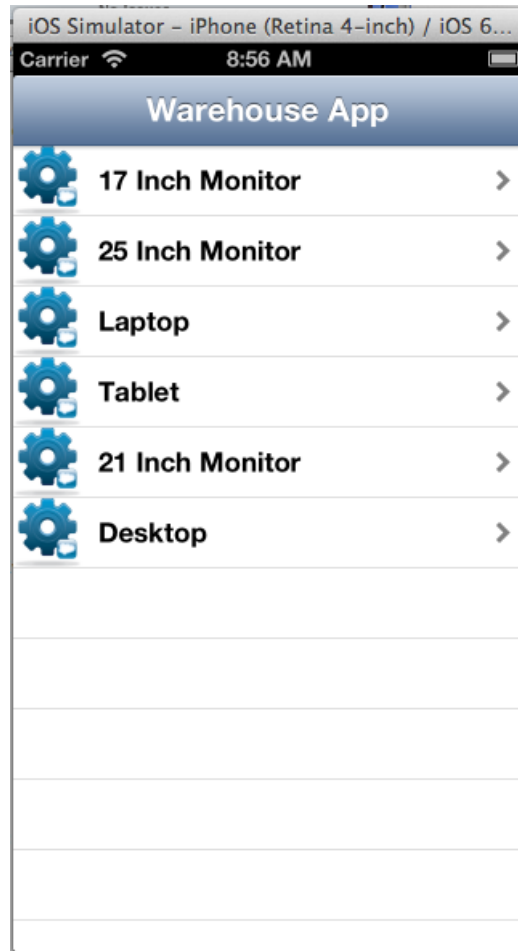
```
SFRestRequest *request = [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name  
FROM User LIMIT 10"];
```

to:

```
SELECT Name, Id, Quantity__c, Price__c FROM Merchandise__c LIMIT 10
```

Step 3: Try Out the App

Build and run the app. When prompted, log into your DE org. The initial page should look similar to the following screen.



At this point, if you click a Merchandise record, nothing happens. You'll fix that in the next tutorial.

Create the Detail Screen

In the previous tutorial, you modified the template app so that, after it starts, it lists up to ten Merchandise records. In this tutorial, you finish the job by creating a detail view and controller. You also establish communication between list view and detail view controllers.

Step 1: Create the App's Detail View Controller

When a user taps a Merchandise record in the Warehouse view, an `IBAction` generates record-specific information and then loads a view from `DetailViewController` that displays this information. However, this view doesn't yet exist, so let's create it.

1. Click **File > New > File...** > **Cocoa Touch > Objective-C class**.
2. Create a new Objective-C class named `DetailViewController` that subclasses `UIViewController`. Make sure that **With XIB for user interface** is checked.
3. Click **Next**.
4. Place the new class in the **Classes** group under **Mobile Warehouse App** in the **Groups** drop-down menu.

Xcode creates three new files in the `Classes` folder: `DetailViewController.h`, `DetailViewController.m`, and `DetailViewController.xib`.

5. Select `DetailViewController.m` in the Project Navigator, and delete the following method declaration:

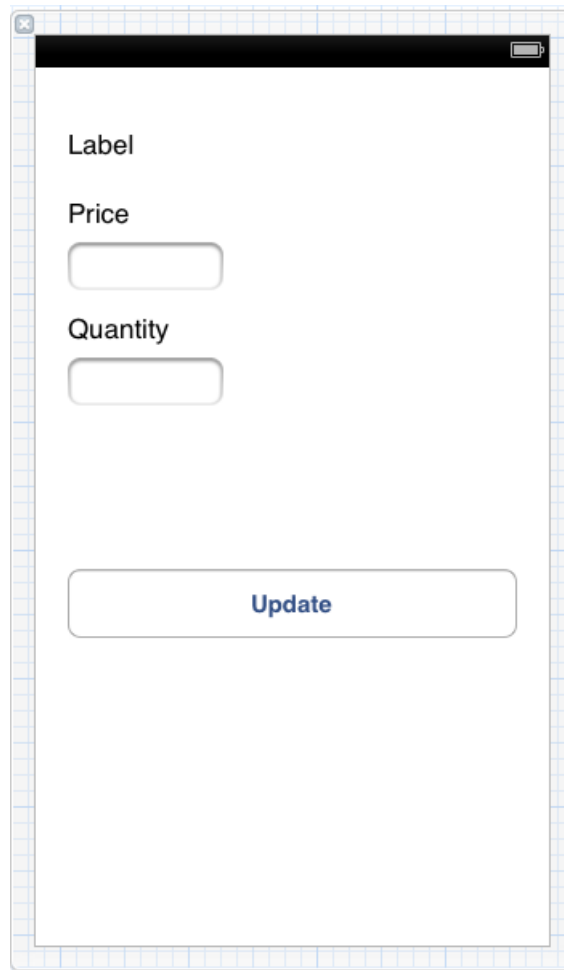
```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization
    }
    return self;
}
```

In this app, you don't need this initialization method because you're not specifying a NIB file or bundle.

6. Select `DetailViewController.xib` in the Project Navigator to open the Interface Builder.

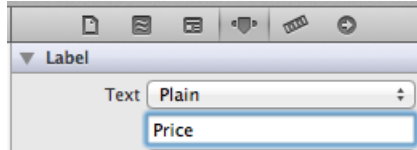


7. From the Utilities view, drag three labels, two text fields, and one button onto the view layout. Arrange and resize the controls so that the screen looks like this:



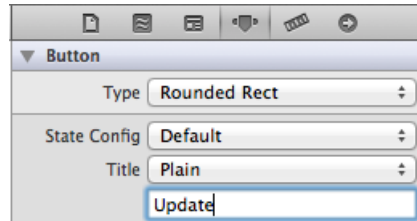
We'll refer to topmost label as the Name label. This label is dynamic. In the next tutorial, you'll add controller code that resets it at runtime to a meaningful value.

8. In the Attributes inspector, set the display text for the static Price and Quantity labels to the values shown. Select each label individually in the Interface Builder and specify display text in the unnamed entry field below the Text drop-down menu.



Note: Adjust the width of the labels as necessary to see the full display text.

9. In the Attributes inspector, set the display text for the Update button to the value shown. Select the button in the Interface Builder and specify its display text in the unnamed entry field below the Title drop-down menu.



10. Build and run to check for errors. You won't yet see your changes.

The detail view design shows Price and Quantity fields, and provides a button for updating the record's Quantity. However, nothing currently works. In the next step, you learn how to connect this design to Warehouse records.

Step 2: Set Up DetailViewController

To establish connections between view elements and their view controller, you can use the Xcode Interface Builder to connect UI elements with code elements.

Add Instance Properties

1. Create properties in `DetailViewController.h` to contain the values passed in by the `WarehouseViewController`: Name, Quantity, Price, and Id. Place these properties within the `@interface` block. Declare each `nonatomic` and `strong`, using these names:

```
@interface DetailViewController : UIViewController

@property (nonatomic, strong) NSNumber *quantityData;
@property (nonatomic, strong) NSNumber *priceData;
@property (nonatomic, strong) NSString *nameData;
@property (nonatomic, strong) NSString *idData;

@end
```

2. In `DetailViewController.m`, just after the `@implementation` tag, synthesize each of the properties.

```
@implementation DetailViewController

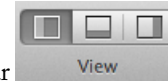
@synthesize nameData;
@synthesize quantityData;
@synthesize priceData;
@synthesize idData;
```

Add IBOutlet Variables

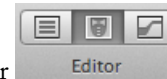
IBOutlet member variables let the controller manage each non-static control. Instead of coding these manually, you can use the Interface Builder to create them. Interface Builder provides an Assistant Editor that gives you the convenience of side-by-side editing windows. To make room for the Assistant Editor, you'll usually want to reclaim screen area by hiding unused controls.

1. In the Project Navigator, click the `DetailViewController.xib` file.

The `DetailViewController.xib` file opens in the Standard Editor.

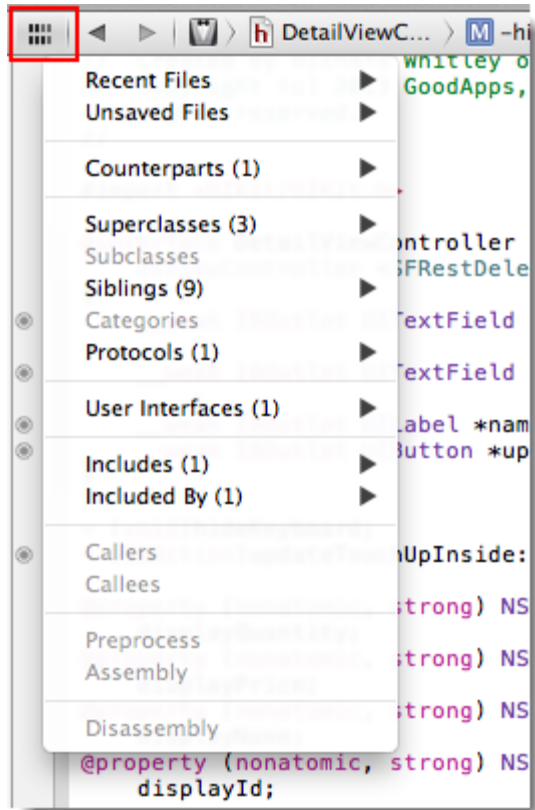


2. Hide the Navigator by clicking Hide or Show Navigator on the View toolbar. Alternatively, you can choose **View > Navigators > Hide Navigators** in the Xcode menu.



3. Open the Assistant Editor by clicking Show the Assistant editor in the Editor toolbar. Alternatively, you can choose **View > Assistant Editor > Show Assistant Editor** in the Xcode menu.

Because you opened `DetailViewController.xib` in the Standard Editor, the Assistant Editor shows the `DetailViewController.h` file. The Assistant Editor guesses which files are most likely to be used together. If you need to open a different file, click the Related Files control in the upper left hand corner of the Assistant Editor



4. At the top of the interface block in `DetailViewController.h`, add a pair of empty curly braces:

```
@interface DetailViewController : UIViewController <SFRestDelegate>
{
}
```

5. In the Standard Editor, control-click the Price text field control and drag it into the new curly brace block in the `DetailViewController.h` file.

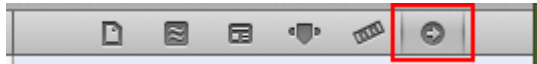
6. In the popup dialog box, name the new outlet `_priceField`, and click **Connect**.
7. Repeat steps 2 and 3 for the Quantity text field, naming its outlet `_quantityField`.
8. Repeat steps 2 and 3 for the Name label, naming its outlet `_nameLabel`.

Your interface code now includes this block:

```
@interface DetailViewController : UIViewController
{
    __weak IBOutlet UITextField *_priceField;
    __weak IBOutlet UITextField *_quantityField;
    __weak IBOutlet UILabel *_nameLabel;
}
```

Add an Update Button Event

1. In the Interface Builder, select the **Update** button and open the Connections Inspector



2. In the Connections Inspector, select the circle next to **Touch Up Inside** and drag it into the `DetailViewController.h` file. Be sure to drop it below the closing curly brace. Name it `updateTouchUpInside`, and click **Connect**.

The Touch Up Inside event tells you that the user raised the finger touching the Update button without first leaving the button. You'll perform a record update every time this notification arrives.

Step 3: Create the Designated Initializer

Now, let's get down to some coding. Start by adding a new initializer method to `DetailViewController` that takes the name, ID, quantity, and price. The method name, by convention, must begin with "init".

1. Click **Show the Standard Editor** and open the Navigator.
2. Add this declaration to the `DetailViewController.h` file just above the `@end` marker:

```
- (id) initWithName:(NSString *)recordName
               objectId:(NSString *)salesforceId
               quantity:(NSNumber *)recordQuantity
               price:(NSNumber *)recordPrice;
```

Later, we'll code `WarehouseViewController` to use this method for passing data to the `DetailViewController`.

3. Open the `DetailViewController.m` file, and copy the signature you created in the previous step to the end of the file, just above the `@end` marker.
4. Replace the terminating semi-colon with a pair of curly braces for your implementation block.

```
- (id) initWithName:(NSString *)recordName
               objectId:(NSString *)salesforceId
               quantity:(NSNumber *)recordQuantity
               price:(NSNumber *)recordPrice {
```

5. In the method body, send an `init` message to the super class. Assign the return value to `self`:

```
self = [super init];
```

This `init` message gives you a functional object with base implementation which will serve as your return value.

6. Add code to verify that the super class initialization succeeded, and, if so, assign the method arguments to the corresponding instance variables. Finally, return `self`.

```
if (self) {
    self.nameData = recordName;
    self.idData = salesforceId;
    self.quantityData = recordQuantity;
    self.priceData = recordPrice;
}
return self;
```

Here's the completed method:

```
- (id) initWithName:(NSString *)recordName
               sobjectId:(NSString *)salesforceId
               quantity:(NSNumber *)recordQuantity
               price:(NSNumber *)recordPrice {
    self = [super init];
    if (self) {
        self.nameData = recordName;
        self.idData = salesforceId;
        self.quantityData = recordQuantity;
        self.priceData = recordPrice;
    }
    return self;
}
```

7. To make sure the controls are updated each time the view appears, add a new `viewWillAppear:` event handler after the `viewDidLoad` method implementation. Begin by calling the super class method.

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
}
```

8. Copy the values of the property variables to the corresponding dynamic controls.

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    [_nameLabel setText:self.nameData];
    [_quantityField setText:[self.quantityData stringValue]];
    [_priceField setText:[self.priceData stringValue]];
}
```

9. Build and run your project to make sure you've coded everything without compilation errors. The app will look the same as it did at first, because you haven't yet added the code to launch the Detail view.



Note: The `[super init]` message used in the `initWithName:` method calls `[super initWithNibName:bundle:]` internally. We use `[super init]` here because we're not passing a NIB name or a bundle. If you are specifying these resources in your own projects, you'll need to call `[super initWithNibName:bundle:]` explicitly.

Step 4: Establish Communication Between the View Controllers

Any view that consumes Salesforce content relies on a `SFRestAPI` delegate to obtain that content. You can designate a single view to be the central delegate for all views in the app, which requires precise communication between the view controllers. For this exercise, let's take a slightly simpler route: Make `WarehouseViewController` and `DetailViewController` each serve as its own `SFRestAPI` delegate.

Update WarehouseViewController

First, let's equip `WarehouseViewController` to pass the quantity and price values for the selected record to the detail view, and then display that view.

1. In `WarehouseViewController.m`, above the `@implementation` block, add the following line:

```
#import "DetailViewController.h"
```

2. On a new line after the `#pragma mark - Table view data source` marker, type the following starter text to bring up a list of `UITableView` delegate methods:

```
- (void)tableView
```

3. From the list, select the `tableView:didSelectRowAtIndexPath:` method.
4. Change the `tableView` parameter name to `itemTableView`.

```
- (void)tableView:(UITableView *)itemTableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
```

5. At the end of the signature, type an opening curly brace (`{`) and press return to stub in the method implementation block.
6. At the top of the method body, per standard iOS coding practices, add the following call to deselect the row.

```
[itemTableView deselectRowAtIndexPath:indexPath animated:NO];
```

7. Next, retrieve a pointer to the `NSDictionary` object associated with the selected data row.

```
NSDictionary *obj = [self.dataRows objectAtIndex:indexPath.row];
```

8. At the end of the method body, create a local instance of `DetailViewController` by calling the `DetailViewController.initWithName:salesforceId:quantity:price:` method. Use the data stored in the `NSDictionary` object to set the name, Salesforce ID, quantity, and price arguments. The finished call looks like this:

```
DetailViewController *detailController =
    [[DetailViewController alloc] initWithName:[obj objectForKey:@"Name"]
                                         salesforceId:[obj objectForKey:@"Id"]
                                         quantity:[obj objectForKey:@"Quantity__c"]
                                         price:[obj objectForKey:@"Price__c"]];
```

9. To display the Detail view, add code that pushes the initialized `DetailViewController` onto the `UINavigationController` stack:

```
[[self navigationController] pushViewController:detailController animated:YES];
```

Great! Now you're using a UINavigationController stack to handle a set of two views. The root view controller is always at the bottom of the stack. To activate any other view, you just push its controller onto the stack. When the view is dismissed, you pop its controller, which brings the view below it back into the display.

10. Build and run your app. Click on any Warehouse item to display its details.

Add Update Functionality

Now that the WarehouseViewController is set up, we need to modify the DetailViewController class to send the user's updates to Salesforce via a REST request.

1. In the DetailViewController.h file, add an instance method to DetailViewController that lets a user update the price and quantity fields. This method needs to send a record ID, the names of the fields to be updated, the new quantity and price values, and the name of the object to be updated. Add this declaration after the interface block and just above the @end marker.

```
- (void)updateWithObjectType:(NSString *)objectType
    objectId:(NSString *)objectId
    quantity:(NSString *)quantity
    price:(NSString *)price;
```

To implement the method, you create an SFRestRequest object using the input values, then send the request object to the shared instance of the SFRestAPI.

2. In the DetailViewController.m file, add the following line above the @implementation block.

```
#import "SFRestAPI.h"
```

3. At the end of the file, just above the @end marker, copy the updateWithObjectType:objectId:quantity:price: signature, followed by a pair of curly braces:

```
- (void)updateWithObjectType:(NSString *)objectType
    objectId:(NSString *)objectId
    quantity:(NSString *)quantity
    price:(NSString *)price {
}
}
```

4. In the implementation block, create a new NSDictionary object to contain the Quantity and Price fields. To allocate this object, use the dictionaryWithObjectsAndKeys: ... NSDictionary class method with the desired list of fields.

```
- (void)updateWithObjectType:(NSString *)objectType
    objectId:(NSString *)objectId
    quantity:(NSString *)quantity
    price:(NSString *)price {

    NSDictionary *fields = [NSDictionary dictionaryWithObjectsAndKeys:
                            quantity, @"Quantity__c",
                            price, @"Price__c",
                            nil];
}
```


5. Create a `SFRestRequest` object. To allocate this object, use the `requestForUpdateWithObjectType:objectId:fields:` instance method on the `SFRestAPI` shared instance.

```
- (void)updateWithObjectType:(NSString *)objectType
    objectId:(NSString *)objectId
    quantity:(NSString *)quantity
    price:(NSString *)price {

    NSDictionary *fields = [NSDictionary dictionaryWithObjectsAndKeys:
                            quantity, @"Quantity__c",
                            price, @"Price__c",
                            nil];

    SFRestRequest *request =
        [[SFRestAPI sharedInstance]
         requestForUpdateWithObjectType:objectType
                             objectId:objectId
                             fields:fields];
}
```

6. Finally, send the new `SFRestRequest` object to the service by calling `send:delegate:` on the `SFRestAPI` shared instance. For the delegate argument, be sure to specify `self`, since `DetailViewController` is the `SFRestDelegate` in this case.

```
- (void)updateWithObjectType:(NSString *)objectType
    objectId:(NSString *)objectId
    quantity:(NSString *)quantity
    price:(NSString *)price {

    NSDictionary *fields = [NSDictionary dictionaryWithObjectsAndKeys:
                            quantity, @"Quantity__c",
                            price, @"Price__c",
                            nil];

    SFRestRequest *request =
        [[SFRestAPI sharedInstance]
         requestForUpdateWithObjectType:objectType
                             objectId:objectId
                             fields:fields];

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

7. Edit the `updateTouchUpInside:` action method to call the `updateWithObjectType:objectId:quantity:price:` method when the user taps the **Update** button.

```
- (IBAction)updateTouchUpInside:(id) sender {
    // For Update button
    [self updateWithObjectType:@"Merchandise__c"
                        objectId:self.idData
                        quantity:[_quantityField text]
                        price:[_priceField text]];
}
```



Note:

- **Extra credit:** Improve your app's efficiency by performing updates only when the user has actually changed the quantity value.

Add SFRestDelegate to DetailViewController

We're almost there! We've issued the REST request, but still need to provide code to handle the response.

1. Open the `DetailViewController.h` file and change the `DetailViewController` interface declaration to include `<SFRestDelegate>`

```
@interface DetailViewController : UIViewController <SFRestDelegate>
```

2. Open the `WarehouseViewController.m` file.
3. Find the pragma that marks the `SFRestAPIDelegate` section.

```
#pragma mark - SFRestAPIDelegate
```

4. Copy the four methods under this pragma into the `DetailViewController.m` file.

```
- (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];
}

- (void)request:(SFRestRequest*)request didFailLoadWithError:(NSError*)error {
    NSLog(@"request:didFailLoadWithError: %@", error);
    //add your failed error handling here
}

- (void)requestDidCancelLoad:(SFRestRequest *)request {
    NSLog(@"requestDidCancelLoad: %@", request);
    //add your failed error handling here
}

- (void)requestDidTimeout:(SFRestRequest *)request {
    NSLog(@"requestDidTimeout: %@", request);
    //add your failed error handling here
}
```

These methods are all we need to implement the `SFRestAPI` interface. For this tutorial, we can retain the simplistic handling of error, cancel, and timeout conditions. However, we need to change the `request:didLoadResponse:` method to suit the detail view purposes. Let's use the `UINavigationController` stack to return to the list view after an update occurs.

5. In the `DetailViewController.m` file, delete the existing code in the `request:didLoadResponse:` delegate method. In its place, add code that logs a success message and then pops back to the root view controller. The revised method looks like this.

```
- (void)request:(SFRestRequest *)request didLoadResponse:(id)jsonResponse {
    NSLog(@"1 record updated");
    [self.navigationController popViewControllerAnimated:YES];
}
```

6. Build and run your app. In the Warehouse view, click one of the items. You're now able to access the Detail view and edit its quantity, but there's a problem: the keyboard won't go away when you want it to. You need to add a little finesse to make the app truly functional.

Hide the Keyboard

The iOS keyboard remains visible as long as any text input control on the screen is responding to touch events. This is where the “First Responder” setting, which you might have noticed in the Interface Builder, comes into play. We didn’t set a first responder because our simple app just uses the default UIKit behavior. As a result, iOS can consider any input control in the view to be the first responder. If none of the controls explicitly tell iOS to hide the keyboard, it remains active.

You can resolve this issue by making every touch-enabled edit control resign as first responder.

1. In `DetailViewController.h`, below the curly brace block, add a new instance method named `hideKeyboard` that takes no arguments and returns `void`.

```
- (void)hideKeyboard;
```

2. In the implementation file, implement this method to send a `resignFirstResponder` message to each touch-enabled edit control in the view.

```
- (void)hideKeyboard {
    [_quantityField resignFirstResponder];
    [_priceField resignFirstResponder];
}
```

The only remaining question is where to call the `hideKeyboard` method. We want the keyboard to go away when the user taps outside of the text input controls. There are many likely events that we could try, but the only one that is sure to catch the background touch under all circumstances is `[UIResponder touchesEnded:withEvent:]`.

3. Since the event is already declared in a class that `DetailViewController` inherits, there’s no need to re-declare it in the `DetailViewController.h` file. Rather, in the `DetailViewController.m` file, type the following incomplete code on a new line outside of any method body:

```
- (void)t
```

A popup menu displays with a list of matching instance methods from the `DetailViewController` class hierarchy.



Note: If the popup menu doesn’t appear, just type the code described next.

4. In the popup menu, highlight the `touchesEnded:withEvent:` method and press Return. The editor types the full method signature into your file for you. Just type an opening brace, press Return, and your stub method is completed by the editor. Within this stub, send a `hideKeyboard` message to `self`.

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event{
    [self hideKeyboard];
}
```

Normally, in an event handler, you’d be expected to call the super class version before adding your own code. As documented in the iOS Developer Library, however, leaving out the super call in this case is a common usage pattern. The only “gotcha” is that you also have to implement the other touches event handlers, which include:

```
- touchesBegan:withEvent:
- touchesMoved:withEvent:
- touchesCancelled:withEvent:
```

The good news is that you only need to provide empty stub implementations.

5. Use the Xcode editor to add these stubs the same way you added the `touchesEnded:` stub. Make sure your final code looks like this:

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event{
    [self hideKeyboard];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event{
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event{
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event{
}
```

Refreshing the Query with `viewWillAppear`

The `viewDidLoad` method lets you configure the view when it first loads. In the `WarehouseViewController` implementation, this method contains the REST API query that populates both the list view and the detail view. However, since `WarehouseViewController` represents the root view, the `viewDidLoad` notification is called only once—when the view is initialized. What does this mean? When a user updates a quantity in the detail view and returns to the list view, the query is not refreshed. Thus, if the user returns to the same record in the detail view, the updated value does not display, and the user is not happy.

You need a different method to handle the query. The `viewWillAppear` method is called each time its view is displayed. Let's add this method to `WarehouseViewController` and move the SOQL query into it.

1. In the `WarehouseViewController.m` file, add the following code after the `viewDidLoad` implementation.

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
}
```

2. Cut the following lines from the `viewDidLoad` method and paste them into the `viewWillAppear:` method, after the call to `super:`

```
    SFRestRequest *request = [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name,
ID,
    Price__c, Quantity__c FROM Merchandise__c LIMIT 10"];
    [[SFRestAPI sharedInstance] send:request delegate:self];
```

The final `viewDidLoad` and `viewWillAppear:` methods look like this.

```
- (void)viewDidLoad{
    [super viewDidLoad];
    self.title = @"Warehouse App";
}

- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    //Here we use a query that should work on either Force.com or Database.com
    SFRestRequest *request = [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name,
ID,
    Price__c, Quantity__c FROM Merchandise__c LIMIT 10"];
```

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

The `viewWillAppear:` method refreshes the query each time the user navigates back to the list view. Later, when the user revisits the detail view, the list view controller updates the detail view with the refreshed data.

Step 5: Try Out the App

1. Build your app and run it in the iPhone emulator. If you did everything correctly, a detail page appears when you click a Merchandise record in the Warehouse screen.
2. Update a record's quantity and price. Be sure to click the **Update** button in the detail view after you edit the values. When you navigate back to the detail view, the updated values display.
3. Log into your DE org and view the record using the browser UI to see the updated values.

iOS Native Sample Applications

The app you created in [Running the Xcode Project Template App](#) is itself a sample application, but it only does one thing: issue a SOQL query and return a result. The native iOS sample apps demonstrate more functionality you can examine and work into your own apps.

- **RestAPIExplorer** exercises all of the native REST API wrappers. It resides in the Mobile SDK for iOS under `native/SampleApps/RestAPIExplorer`.
- **NativeSqlAggregator** shows SQL aggregation examples as well as a native SmartStore implementation. It resides in the Mobile SDK for iOS under `native/SampleApps/NativeSqlAggregator`.
- **FileExplorer** demonstrates the Files API as well as the underlying MKNetwork network enhancements. It resides in the Mobile SDK for iOS under `native/SampleApps/FileExplorer`.

Chapter 5

Native Android Development

In this chapter ...

- [Android Native Quick Start](#)
- [Native Android Requirements](#)
- [Creating an Android Project](#)
- [Setting Up Sample Projects in Eclipse](#)
- [Developing a Native Android App](#)
- [Tutorial: Creating a Native Android Warehouse Application](#)
- [Android Native Sample Applications](#)

Salesforce Mobile SDK delivers libraries and sample projects for developing native mobile apps on Android.

The Android native SDK provides two main features:

- Automation of the OAuth2 login process, making it easy to integrate the process with your app.
- Access to the Salesforce REST API, with utility classes that simplify that access.

The Android Salesforce Mobile SDK includes several sample native applications. It also provides an `ant` target for quickly creating a new application.

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 the [Mobile SDK for Android](#).
3. At the command line, run the forcedroid application to create a new [Android project](#), and then run that app in Eclipse or from the command line.
4. Set up [sample projects in Eclipse](#).

Native Android Requirements

Mobile SDK 2.2 Android development requires the following software.

- Java JDK 6 or higher—<http://www.oracle.com/downloads>.
- Apache Ant 1.8 or later—<http://ant.apache.org>.
- Android SDK Tools, version 21 or later—<http://developer.android.com/sdk/installing.html>.



Note: For best results, install all previous versions of the Android SDK as well as your target version.

- Eclipse—<https://www.eclipse.org>. Check the [Android Development Tools website](#) for the minimum supported Eclipse version.
- Android ADT (Android Development Tools) plugin for Eclipse, version 21 or later—<http://developer.android.com/sdk>.
- In order to run the application in the Emulator, you need to set up at least one Android Virtual Device (AVD) that targets Platform 2.2 or above (we recommend 4.0 or above). To learn how to set up an AVD in Eclipse, follow the instructions at <http://developer.android.com/guide/developing/devices/managing-avds.html>.

On the Salesforce side, you'll also need:

- Salesforce Mobile SDK 2.2 for Android. See [Install the Mobile SDK](#).
- A Salesforce [Developer Edition organization](#) with a [connected app](#).

The `SalesforceSDK` project is built with the Android 3.0 (Honeycomb) library. The primary reason for this is that we want to be able to make a conditional check at runtime for file system encryption capabilities. This check is bypassed on earlier Android platforms; thus, you can still use the `salesforcesdk.jar` in earlier Android application versions, down to the minimum-supported Android 2.2.

Creating an Android Project

To create a new app, use forcedroid again on the command line. You have two options for configuring your app.

- Configure your application options interactively as prompted by the forcedroid app.
- Specify your application options directly at the command line.

Specifying Application Options Interactively

To enter application options interactively, do one of the following:

- If you installed Mobile SDK globally, type `forcedroid create`.
- If you installed Mobile SDK locally, type `<forcedroid_path>/node_modules/.bin/forcedroid create`.

The forcedroid utility prompts you for each configuration option.

```
rwhitley-ltm1:Downloads rwhitley$ forcedroid create
Enter your application type (native, hybrid_remote, or hybrid_local): native
Enter your application name: MyNativeAndroidApp
Enter the target directory of your app: /Users/rwhitley/Development/AndroidApps
Enter the package name for your app (com.mycompany.my_app): com.acme.goodapps
Do you want to use SmartStore in your app? [yes/NO] ('No' by default)
Adjusting SalesforceSDK library project reference in project.properties.
Renaming application class to MyNativeAndroidAppApp in source.
Renaming application to MyNativeAndroidApp in source.
Renaming package name to com.acme.goodapps in source.
Moving source files to proper package path.
Renaming the app class filename to MyNativeAndroidAppApp.java.

Your application project is ready in /Users/rwhitley/Development/AndroidApps.
```

Specifying Application Options Directly

You can also specify your configuration directly by typing the full forcedroid command string. To see usage information, type `forcedroid` without arguments. The list of available options displays:

```
$ node_modules/.bin/forcedroid
Usage:
forcedroid create
  --apptype=<Application Type> (native, hybrid_remote, hybrid_local)
  --appname=<Application Name>
  --targetdir=<Target App Folder>
  --packagename=<App Package Identifier> (com.my_company.my_app)
  --startpage=<Path to the remote start page> (/apex/MyPage - Only required/used for
'hybrid_remote')
  [--usesmartstore=<Whether or not to use SmartStore> ('true' or 'false', false by
default)]
```

Using this information, type `forcedroid create`, followed by your options and values. For example:

```
$ node_modules/.bin/forcedroid create --apptype="native" --appname="packagetest"
--targetdir="PackageTest" --packagename="com.test.my_new_app"
```

Importing and Building Your App in Eclipse

Use the following instructions to build and run your new app in the Eclipse editor.

1. Launch Eclipse and select your target directory as the workspace directory.
2. Select **Eclipse > Preferences**, choose the **Android** section, and enter the Android SDK location.
3. Click OK.
4. Select **File > Import** and select **General > Existing Projects into Workspace**.
5. Click Next.
6. Specify the `forcedroid/native` directory as your root directory. Next to the list that displays, click **Deselect All**, then browse the list and check the **SalesforceSDK** project.
7. If you set `-use_smartstore=true`, check the **SmartStore** project as well.
8. Click **Import**.
9. Repeat Steps 4–8. In Step 6, choose your target directory as the root, then select only your new project.

When you've finished importing the projects, Eclipse automatically builds your workspace. This process can take several minutes. When the status bar reports zero errors, you're ready to run the project.

1. In your Eclipse workspace, Control-click or right-click your project.
2. From the popup menu, choose **Run As > Android Application**.

Eclipse 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 only if you want to build and run your app from the command line.

1. Before building the new application, build the SalesforceSDK project by running the following commands at the command prompt:

```
cd $SALESFORCE_SDK_DIR/native/SalesforceSDK
$ANDROID_SDK_DIR/tools/android update project -p . -t <id>
ant clean debug
```

where `SALESFORCE_SDK_DIR` points to your Salesforce SDK installation directory, and `ANDROID_SDK_DIR` points to your Android SDK directory.



Note: The `-t <id>` parameter specifies API level of the target Android version. Use `android.bat list targets` to see the IDs for API versions installed on your system. See [Native Android Requirements](#) for supported API levels.

2. Build the SmartStore project by running the following commands at the command prompt:

```
cd $SALESFORCE_SDK_DIR/hybrid/SmartStore
$ANDROID_SDK_DIR/tools/android update project -p . -t <id>
ant clean debug
```

where `SALESFORCE_SDK_DIR` points to your Salesforce SDK installation directory, and `ANDROID_SDK_DIR` points to your Android SDK directory.

3. To build the new application, run the following commands at the command prompt:

```
cd <your_project_directory>
$ANDROID_SDK_DIR/tools/android update project -p . -t <id>
ant clean debug
```

where `ANDROID_SDK_DIR` points to your Android SDK directory.

4. If your emulator is not running, use the Android AVD Manager to start it. If you're using a device, connect it.
5. Type the following command at the command prompt:

```
ant installld
```



Note: You can safely ignore the following warning:

```
It seems that there are sub-projects. If you want to update them please use the
--subprojects parameter.
```

The Android project you created contains a simple application you can build and run.

forcedroid Command Parameters

The following table describes the forcedroid command parameters.

Parameter Name	Description
--apptype	One of the following: <ul style="list-style-type: none"> “native” “hybrid_remote” (server-side hybrid app using VisualForce) “hybrid_local” (client-side hybrid app that doesn’t use VisualForce)
--appname	Name of your application
--targetdir	Folder in which you want your project to be created. If the folder doesn’t exist, the script creates it.
--packagename	Package identifier for your application (for example, “com.acme.app”).
--apexpage	(hybrid remote apps only) Server path to the Apex start page. For example: /apex/MyAppStartPage.
--usesmartstore=true	(Optional) Include only if you want to use SmartStore for offline data. Defaults to false if not specified.

Setting Up Sample Projects in Eclipse

The repository you cloned has other sample apps you can run. To import those into Eclipse:

1. Launch Eclipse and select `--target_dir` as your workspace directory.
2. If you haven’t done so already, select **Window > Preferences**, choose the **Android** section, and enter the Android SDK location. Click OK.
3. Select **File > Import** and select **General > Existing Projects into Workspace**.
4. Click Next.
5. Select `forcedroid/native` as your root directory and import the projects listed in [Android Project Files](#).

Android Project Files

Inside the `$NATIVE_DIR`, you will find several projects:

- `SalesforceSDK`—Salesforce Mobile SDK project. Provides support for OAuth2 and REST API calls
- `test/SalesforceSDKTest`—App for testing the SalesforceSDK project
- `TemplateApp`—App used as a template when creating new native applications using Mobile SDK
- `test/TemplateAppTest`—App for testing the TemplateApp project
- `SampleApps/RestExplorer`—Sample app using SalesforceSDK to explore the REST API calls
- `SampleApps/FileExplorer`—Sample app that demonstrates the Files API
- `SampleApps/NativeSqlAggregator` —Sample native app that uses SmartStore

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 Java and Android development well enough to write code that uses Mobile SDK native classes.

The `create_native` Script

If you manually installed Mobile SDK from GitHub, use the `create_native` script, instead of `forcedroid`, to create a new native project. The `create_native` script creates the app folder you specify, then populates it with a project file, build file, manifest file and resource files. Next, it copies the entire `TemplateApp` project to the new folder. It then updates the project properties, file names, class names, and directory paths to match the new app's configuration. As a result, your new project replicates all the settings and components used by the `TemplateApp` project.

If your new app supports SmartStore, the script also:

- Adds the SmartStore support library to the app directory.
- References the SmartStore library in the new project's properties.
- Changes the application class to extend `SalesforceSDKManagerWithSmartStore` rather than `SalesforceSDKManager`.

Finally, the script posts an important message:

```
"Before you ship, make sure to plug in your oauth client id and callback url in:
    ${target.dir}/res/values/bootconfig.xml"
```

If you're wondering where to get the OAuth client ID and callback URL, look in your connected app definition in your Salesforce organization. The OAuth client ID is the connected app's Consumer Key. The callback URL is the one you specified when you created your connected app. You enter these keys in the `res/values/bootconfig.xml` file of your project, which contains a few clearly named `<string>` nodes. Here's an example `bootconfig.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
    <string name="remoteAccessConsumerKey">3MVG92.uWdyphVj4bnold7yuIpCQsNgddW
        tqRND3faxrv9uKnbj47H4RkweHA2lKY4cBusvDVp0M6gdGE8hp</string>
    <string name="oauthRedirectURI">sfdc:///axm/detect/oauth/done</string>
    <string-array name="oauthScopes">
        <item>api</item>
    </string-array>
</resources>
```

The `create_native` script pre-populates `oauthRedirectURI` and `remoteAccessConsumerKey` strings with dummy values. Replace those values with the strings from your connected app definition.

Android Application Structure

Typically, native Android apps that use the Mobile SDK require:

- An application entry point class that extends `android.app.Application`.
- At least one activity that extends `android.app.Activity`.

With the Mobile SDK, you:

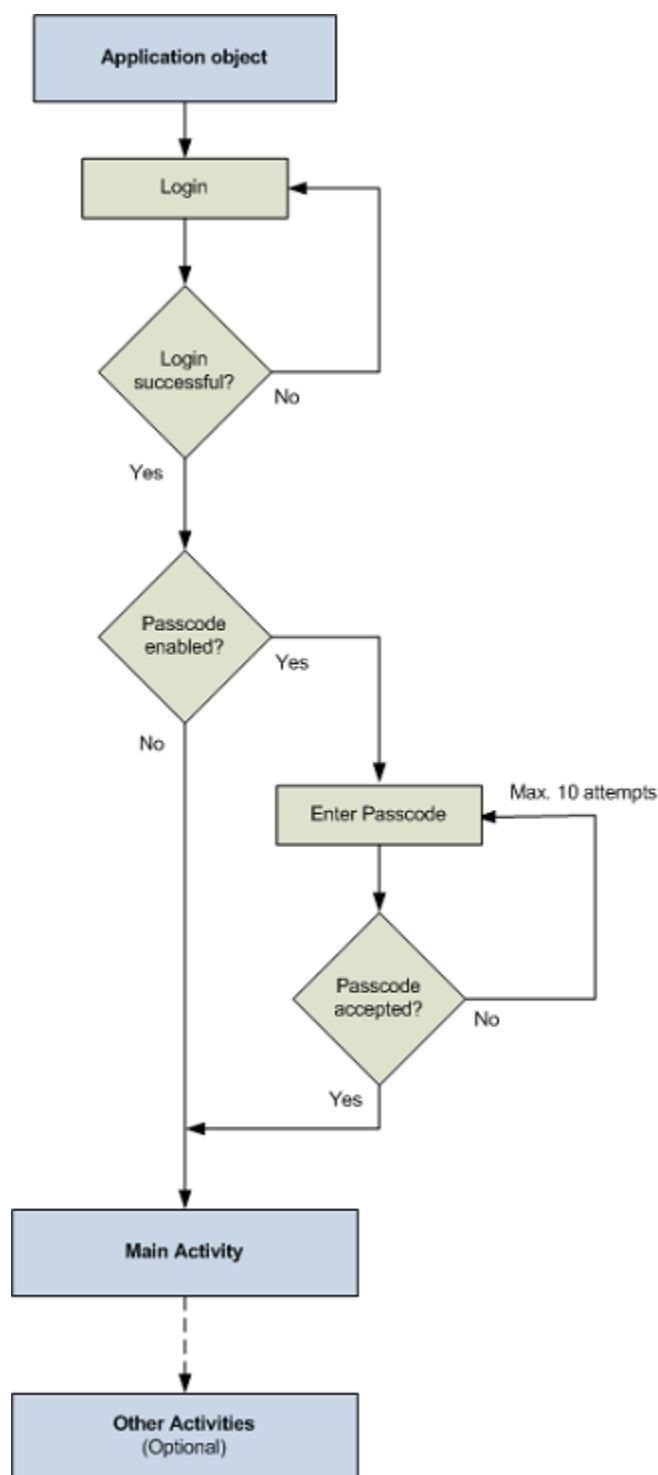
- Create a stub class that extends `android.app.Application`.
- Implement `onCreate()` in your `Application` stub class to call `SalesforceSDKManager.initNative()`.
- Extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`. This extension is optional but recommended.

The top-level `SalesforceSDKManager` class implements passcode functionality for apps that use passcodes, and fills in the blanks for those that don't. It also 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 free handling of application pause and resume events and related passcode management. 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 pause and resume protocols found in `SalesforceActivity`.

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

This overview of the Mobile SDK native classes give you a look at pertinent details of each class and 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 the 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

initNative() Method

During startup, you initialize the singleton `SalesforceSDKManager` object by calling its static `initNative()` method. This method takes four arguments:

Parameter Name	Description
<code>applicationContext</code>	An instance of <code>Context</code> that describes your application's context. In an <code>Application</code> extension class, you can satisfy this parameter by passing a call to <code>getApplicationContext()</code> .
<code>keyImplementation</code>	An instance of your implementation of the <code>KeyInterface</code> Mobile SDK interface. You are required to implement this interface.
<code>mainActivity</code>	The descriptor of the class that displays your main activity. The main activity is the first activity that displays after login.
<code>loginActivity</code>	(Optional) The class descriptor of your custom <code>LoginActivity</code> class.

Here's an example from the `TemplateApp`:

```
SalesforceSDKManager.initNative(getApplicationContext(), new KeyImpl(), MainActivity.class);
```

In this example, `KeyImpl` is the app's implementation of `KeyInterface`. `MainActivity` subclasses `SalesforceActivity` and is designated here as the first activity to be called after login.

logout() Method

The `SalesforceSDKManager.logout()` method clears user data. For example, if you've introduced your own resources that are user-specific, you don't want them to persist into the next user session. `SmartStore` destroys user data and account information automatically at logout.

Always call the superclass method somewhere in your method override, preferably after doing your own cleanup. Here's a pseudo-code example.

```
@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

The Mobile SDK builds a user agent string to publish the app's versioning information at runtime. This user agent takes the following form.

```
SalesforceMobileSDK/<salesforceSDK version> android/<android OS version> appName/appVersion
<Native|Hybrid>
```

Here's a real-world example.

```
SalesforceMobileSDK/2.0 android mobile/4.2 RestExplorer/1.0 Native
```

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.

KeyInterface Interface

`KeyInterface` is a required interface that you implement and pass into the `SalesforceSDKManager.initNative()` method.

getKey() Method

You are required to return a Base64-encoded encryption key from the `getKey()` abstract method. Use the `Encryptor.hash()` and `Encryptor.isBase64Encoded()` helper methods to generate suitable keys. The Mobile SDK uses your key to encrypt app data and account information.

AccountWatcher Class

`AccountWatcher` informs your app when the user's account is removed through Settings. Without `AccountWatcher`, the application gets no notification of these changes. It's important to know when an account is removed so that its passcode and data can be disposed of properly, and logout can begin.

`AccountWatcher` defines an internal interface, `AccountRemoved`, that each app must implement. `SalesforceSDKManager` implements this interface to terminate the app's current (front) activity and reset the passcode, if used, and encryption key.

PasscodeManager Class

The `PasscodeManager` class manages passcode encryption and displays the passcode page as required. It also reads mobile policies and caches them locally. This class is used internally to handle all passcode-related activities with minimal coding on your part. As a rule, apps call only these three `PasscodeManager` methods:

- `public void onPause(Activity ctx)`
- `public boolean onResume(Activity ctx)`
- `public void recordUserInteraction()`

These methods must be called in any native activity class that

- Is in an app that requires a passcode, and
- Does not extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`.

You get this implementation for free in any activity that extends `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`.

`onPause()` and `onResume()`

These methods handle the passcode dialog box when a user pauses and resumes the app. Call each of these methods in the matching methods of your activity class. For example, `SalesforceActivity.onPause()` calls `PasscodeManager.onPause()`, passing in its own class descriptor as the argument, before calling the superclass.

```
@Override
public void onPause() {
    passcodeManager.onPause(this);
    super.onPause();
}
```

Use the boolean return value of `PasscodeManager.onResume()` method as a condition for resuming other actions. In your app's `onResume()` implementation, be sure to call the superclass method before calling the `PasscodeManager` version. For example:

```
@Override
public void onResume() {
    super.onResume();
    // Bring up passcode screen if needed
    passcodeManager.onResume(this);
}
```

`recordUserInteraction()`

This method saves the time stamp of the most recent user interaction. Call `PasscodeManager.recordUserInteraction()` in the activity's `onUserInteraction()` method. For example:

```
@Override
public void onUserInteraction() {
    passcodeManager.recordUserInteraction();
}
```


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 classes provides a free implementation of `PasscodeManager` calls. When possible, it's a good idea to extend one of these classes for all of your app's activities, even if your app doesn't currently use passcodes.

For passcode-protected apps: If any of your activities don't extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`, you'll need to add a bit of passcode protocol to each of those activities. See [Using Passcodes](#)

Each of these activity classes contain a single abstract method:

```
public abstract void onResume(RestClient client);
```

This method overloads the `Activity.onResume()` method, which is implemented by the class. The class method calls your overload after it instantiates a `RestClient` instance. Use this method to cache the client that's passed in, and then use that client to perform your REST requests.

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. With the exceptions of `SalesforceActivity`, `SalesforceListActivity`, and `SalesforceExpandableListActivity`, do not override these activity classes with intentions of replacing the resources at runtime.

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 `WrappedRestRequest`. It then adds the `WrappedRestRequest` object to the request queue and returns that object. If you wish to cancel the request while it's pending, call `cancel()` on the `WrappedRestRequest` object.

getRequestQueue() Method

You can access the underlying `RequestQueue` object by calling `restClient.getRequestQueue()` on your `RestClient` instance. With the `RequestQueue` object you can directly cancel and otherwise manipulate pending requests. For example, you can cancel an entire pending request queue by calling `restClient.getRequestQueue().cancelAll()`. See a code example at [Managing the Request Queue](#).

RestRequest Class

The `RestRequest` class creates and formats REST API requests from the data your app provides. It is implemented by Mobile SDK and serves as a factory for instances of itself.

Don't directly create instances of `RestRequest`. Instead, call an appropriate `RestRequest` static factory method such as `RestRequest.getRequestForCreate()`. To send the request, pass the returned `RestRequest` object to `RestClient.sendAsync()` or `RestClient.sendSync()`. See [Using REST APIs](#).

The `RestRequest` class natively handles the standard Salesforce data operations offered by the Salesforce REST API and SOAP API. Supported operations are:

Operation	Parameters	Description
Versions	None	Returns Salesforce version metadata
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

Operation	Parameters	Description
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
Retrieve	API version, object type, object ID, list of fields	Retrieves a record by object ID
Update	API version, object type, object ID, map of field names to value objects	Updates an object with the given map
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
Delete	API version, object type, object ID	Deletes the object of the given type with the given ID

To obtain an appropriate `RestRequest` instance, call the `RestRequest` static method that matches the operation you want to perform. Here are the `RestRequest` static methods.

- `getRequestForCreate()`
- `getRequestForDelete()`
- `getRequestForDescribe()`
- `getRequestForDescribeGlobal()`
- `getRequestForMetadata()`
- `getRequestForQuery()`
- `getRequestForResources()`
- `getRequestForRetrieve()`
- `getRequestForSearch()`
- `getRequestForUpdate()`
- `getRequestForUpsert()`
- `getRequestForVersions()`

These methods return a `RestRequest` object which you pass to an instance of `RestClient`. The `RestClient` class provides synchronous and asynchronous methods for sending requests: `sendSync()` and `sendAsync()`. Use `sendAsync()` when you're sending a request from a UI thread. Use `sendSync()` only on non-UI threads, such as a service or a worker thread spawned by an activity.

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()`:

```
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 [FileRequests Methods \(Android\)](#). For a full description of the REST request and response bodies, go to **Chatter REST API Resources > Files Resources** at <http://www.salesforce.com/us/developer/docs/chatterapi>.

Method Name	Description
<code>ownedFilesList</code>	Builds a request that fetches a page from the list of files owned by the specified user.
<code>filesInUsersGroups</code>	Builds a request that fetches a page from the list of files owned by the user's groups.
<code>filesSharedWithUser</code>	Builds a request that fetches a page from the list of files that have been shared with the user.
<code>fileDetails</code>	Builds a request that fetches the file details of a particular version of a file.
<code>batchFileDetails</code>	Builds a request that fetches the latest file details of one or more files in a single request.
<code>fileRendition</code>	Builds a request that fetches the a preview/rendition of a particular page of the file (and version).
<code>fileContents</code>	Builds a request that fetches the actual binary file contents of this particular file.
<code>fileShares</code>	Builds a request that fetches a page from the list of entities that this file is shared to.
<code>addFileShare</code>	Builds a request that add a file share for the specified file ID to the specified entity ID.
<code>deleteFileShare</code>	Builds a request that deletes the specified file share.
<code>uploadFile</code>	Builds a request that uploads a new file to the server. Creates a new file.

WrappedRestRequest Class

The `WrappedRestRequest` class subclasses the Volley Request class. You don't create `WrappedRestRequest` objects. The `RestClient.sendAsync()` method uses this class to wrap the `RestRequest` object that you passed in and returns it to the caller. You can use this returned object to cancel the request “in flight” by calling the `cancel()` method.

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 **Pick Server**. **Pick Server** launches an instance of the `ServerPickerActivity` class, which displays **Production**, **Sandbox**, and **Custom Server** options. When a user chooses **Custom Server**, `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

Several other classes in the `ui` package are worth mentioning, although they don't affect your native API development efforts.

The `PasscodeActivity` class provides the UI for the passcode screen. It runs in one of three modes: Create, CreateConfirm, and Check. Create mode is presented the first time a user attempts to log in. It prompts the user to create a passcode. After the user submits the passcode, the screen returns in CreateConfirm mode, asking the user to confirm the new passcode. Thereafter, that user sees the screen in Check mode, which simply requires the user to enter the passcode.

`SalesforceR` is a deprecated class. This class was required when the Mobile SDK was delivered in JAR format, to allow developers to edit resources in the binary file. Now that the Mobile SDK is available as a library project, `SalesforceR` is not needed. Instead, you can override resources in the SDK with your own.

`SalesforceDroidGapActivity` and `SalesforceGapViewClient` are used only in hybrid apps.

UpgradeManager Class

`UpgradeManager` provides a mechanism for silently upgrading the SDK version installed on a device. This class stores the SDK version information in a shared preferences file on the device. To perform an upgrade, `UpgradeManager` queries the current `SalesforceSDKManager` instance for its SDK version and compares its version to the device's version information. If an upgrade is necessary—for example, if there are changes to a database schema or to encryption patterns—`UpgradeManager` can take the necessary steps to upgrade SDK components on the device. This class is intended for future use. Its implementation in Mobile SDK 2.0 simply stores and compares the version string.

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
<code>EventsObservable</code>	See the source code for a list of all events that the Mobile SDK for Android propagates.
<code>EventsObserver</code>	Implement this interface to eavesdrop on any event. This functionality is useful if you're doing something special when certain types of events occur.

Class	Description
<code>TokenRevocationReceiver</code>	This class handles what happens when an administrator revokes a user's refresh token. See Handling Refresh Token Revocation in Android Native Apps .
<code>UriFragmentParser</code>	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 plugins that access native code. The base class for these Mobile SDK plugins is `ForcePlugin`. If you want to implement your own Javascript plugin 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 plugin with `CordovaPlugin`, then you create a Javascript file that mirrors it. Cordova calls the plugin's `execute()` function when a script calls one of the plugin's Javascript functions.

Using Passcodes

User data in Mobile SDK apps is secured by encryption. The administrator of your Salesforce org has the option of requiring the user to enter a passcode for connected apps. In this case, your app uses that passcode as an encryption hash key. If the Salesforce administrator doesn't require a passcode, you're responsible for providing your own key.

Salesforce Mobile SDK does all the work of implementing the passcode workflow. It calls the passcode manager to obtain the user input, and then combines the passcode with prefix and suffix strings into a hash for encrypting the user's data. It also handles decrypting and re-encrypting data when the passcode changes. If an organization changes its passcode requirement, the Mobile SDK detects the change at the next login and reacts accordingly. If you choose to use a passcode, your only responsibility is to implement the `SalesforceSDKManager.getKey()` method. All your implementation has to do in this case is return a Base64-encoded string that can be used as an encryption key.

Internally, passcodes are stored as Base64-encoded strings. The SDK uses the `Encryptor` class for creating hashes from passcodes. You should also use this class to generate a hash when you provide a key instead of a passcode. Passcodes and keys are used to encrypt and decrypt `SmartStore` data as well as `oAuth` tokens, user identification strings, and related security information. To see exactly what security data is encrypted with passcodes, browse the `ClientManager.changePasscode()` method.

Mobile policy defines certain passcode attributes, such as the length of the passcode and the timing of the passcode dialog. Mobile policy files for connected apps live on the Salesforce server. If a user enters an incorrect passcode more than ten consecutive times, the user is logged out. The Mobile SDK provides feedback when the user enters an incorrect passcode, apprising the user of how many more attempts are allowed. Before the screen is locked, the `PasscodeManager` class stores a reference to the front activity so that the same activity can be resumed if the screen is unlocked.

If you define activities that don't extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity` in a passcode-protected app, be sure to call these three `PasscodeManager` methods from each of those activity classes:

- `PasscodeManager.onPause()`
- `PasscodeManager.onResume(Activity)`

- `PasscodeManager.recordUserInteraction()`

Call `onPause()` and `onResume()` from your activity's methods of the same name. Call `recordUserInteraction()` from your activity's `onUserInteraction()` method. Pass your activity class descriptor to `onResume()`. These calls ensure that your app enforces passcode security during these events. See [PasscodeManager Class](#).



Note: The `SalesforceActivity`, `SalesforceListActivity`, and `SalesforceExpandableListActivity` classes implement these mandatory methods for you for free. Whenever possible, base your activity classes on one of these classes.

Resource Handling

Salesforce Mobile SDK resources are configured in XML files that reside in the `native/SalesforceSDK/res` folder. You can customize many of these resources by making changes in this folder.

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
- Values—Strings, colors, and dimensions that are used by the SDK

Two additional resource types are mostly for internal use:

- Menus
- XML

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
<code>drawable</code>	Generic versions of drawable resources
<code>drawable-hdpi</code>	High resolution; for most smart phones
<code>drawable-ldpi</code>	Low resolution; for low-end feature phones
<code>drawable-mdpi</code>	Medium resolution; for low-end smart phones
<code>drawable-xhdpi</code>	Resources for extra high-density screens (~320dpi)
<code>drawable-xlarge</code>	For tablet screens in landscape orientation
<code>drawable-xlarge-port</code>	For tablet screens in portrait orientation
<code>drawable-xxhdpi-port</code>	Resources for extra-extra high density screens (~480 dpi)
<code>layout</code>	Generic versions of layouts
<code>layout-land</code>	For landscape orientation
<code>layout-xlarge</code>	For tablet screens

Folder name	Usage
menus	Add Connection dialog and login menu for phones
values	Generic styles and values
values-xlarge	For tablet screens
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 `EditText` node from `res/layout/sf__passcode.xml` describes a text edit control:

```
<EditText android:id="@+id/sf__passcode_text"
    style="@style/SalesforceSDK.Passcode.Text.Entry"
    android:inputType="textPassword" />
```

In this case, the `EditText` control uses an `android:inputType` attribute. Its value, `"textPassword"`, tells the operating system to obfuscate the typed input.

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.Entry` refers to an SDK-owned style defined in `res/values/sf__styles.xml`. Here's the style definition.

```
<style name="SalesforceSDK.Passcode.Text.Entry">
    <item name="android:layout_width">wrap_content</item>
    <item name="android:lines">1</item>
    <item name="android:maxLength">10</item>
    <item name="android:minWidth">@dimen/sf__passcode_text_min_width</item>
    <item name="android:imeOptions">actionGo</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. 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`.

File name	Contains
<code>sf__colors.xml</code>	Colors referenced by Mobile SDK styles

File name	Contains
<code>sf__dimens.xml</code>	Dimensions referenced by Mobile SDK styles
<code>sf__strings.xml</code>	Strings referenced by Mobile SDK styles; error messages can be overridden
<code>sf__styles.xml</code>	Visual styles used by the Mobile SDK
<code>strings.xml</code>	App-defined strings

You can override the values in `strings.xml`. However, if you used the `create_native` script to create your app, strings in `strings.xml` already reflect appropriate values.

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 plugins for hybrid apps.

Using REST APIs

To query, describe, create, or update data from a Salesforce org, native apps call Salesforce REST APIs. Salesforce REST APIs honor SOQL strings and can accept and return data in either JSON or XML format. REST APIs are fully documented at [REST API Developer's Guide](#). You can find links to related Salesforce development documentation at the [Force.com developer documentation website](#).

With Android native apps, you do only 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 include:

- `ClientManager`—Serves as a factory for `RestClient` instances. It also handles account logins and handshakes with the Salesforce server. Implemented by the Mobile SDK.
- `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. Implemented by the Mobile SDK.
- `RestRequest`—Formats REST API requests from the data your app provides. Also serves as a factory for instances of itself. Don't directly create instances of `RestRequest`. Instead, call an appropriate `RestRequest` static getter function such as `RestRequest.getRequestForCreate()`. Implemented by the SDK.
- `RestResponse`—Formats the response content in the requested format, returns the formatted response to your app, and closes the content stream. The `RestRequest` class creates instances of `RestResponse` and returns them to your app through your implementation of the `RestClient.AsyncRequestCallback` interface. Implemented by the SDK.

The `RestRequest` class natively handles the standard Salesforce data operations offered by the Salesforce REST and SOAP APIs. Supported operations are:

Operation	Parameters	Description
Versions	None	Returns Salesforce version metadata

Operation	Parameters	Description
Resources	API version	Returns available resources for the specified API version, including resource name and URI
Metadata	API version, object type	
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
Retrieve	API version, object type, object ID, list of fields	Retrieves a record by object ID
Update	API version, object type, object ID, map of field names to value objects	Updates an object with the given map
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
Delete	API version, object type, object ID	Deletes the object of the given type with the given ID

To obtain an appropriate `RestRequest` instance, call the `RestRequest` static method that matches the operation you want to perform. Here are the `RestRequest` static methods.

- `getRequestForCreate()`
- `getRequestForDelete()`
- `getRequestForDescribe()`
- `getRequestForDescribeGlobal()`
- `getRequestForMetadata()`
- `getRequestForQuery()`
- `getRequestForResources()`
- `getRequestForRetrieve()`
- `getRequestForSearch()`
- `getRequestForUpdate()`
- `getRequestForUpsert()`
- `getRequestForVersions()`

These methods return a `RestRequest` object which you pass to an instance of `RestClient`. The `RestClient` class provides synchronous and asynchronous methods for sending requests: `sendSync()` and `sendAsync()`. Use `sendAsync()` when you're sending a request from a UI thread. Use `sendSync()` only on non-UI threads, such as a service or a worker thread spawned by an activity.

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. This code from the `native/SampleApps/RestExplorer` sample app implements and instantiates `RestClientCallback` inline:

```
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(ExplorerActivity.this);
            return;
        }
        // Cache the returned client
        ExplorerActivity.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:

```
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.

The following code implements and instantiates `AsyncRequestCallback` inline:

```
private void sendFromUiThread(RestRequest restRequest) {
    client.sendAsync(restRequest, new AsyncRequestCallback() {
        private long start = System.nanoTime();
        @Override
        public void onSuccess(RestRequest request, RestResponse result) {
            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.

Android Template App: Deep Dive

The `TemplateApp` sample project implements everything you need to create a basic Android app. Because it's a "bare bones" example, it also serves as the template that the Mobile SDK's `create_native` ant script uses to set up new native Android projects. You can gain a quick understanding of the native Android SDK by studying this project.

The `TemplateApp` project defines two classes, `TemplateApp` and `MainActivity`. The `TemplateApp` class extends `Application` and calls `SalesforceSDKManager.initNative()` in its `onCreate()` override. The `MainActivity` class subclasses the `SalesforceActivity` class. These two classes are all you need to create a running mobile app that displays a login screen and a home screen.

Despite containing only about 200 lines of code, `TemplateApp` is more than just a "Hello World" example. In its main activity, it retrieves Salesforce data through REST requests and displays the results on a mobile page. You can extend `TemplateApp` by adding more activities, calling other components, and doing anything else that the Android operating system, the device, and security restraints allow.

TemplateApp Class

Every native Android app requires an instance of `android.app.Application`. Here's the entire class:

```
package com.salesforce.samples.templateapp;

import android.app.Application;

import com.salesforce.androidsdk.app.SalesforceSDKManager;

/**
 * Application class for our application.
 */
public class TemplateApp extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        SalesforceSDKManager.initNative(getApplicationContext(), new KeyImpl(),
            MainActivity.class);
    }
}
```

The `TemplateApp` class accomplishes two main tasks:

- Calls `initNative()` to initialize the app
- Passes in the app's implementation of `KeyInterface`

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 MainActivity class for the Template app extends

`com.salesforce.androidsdk.ui.sfnative.SalesforceActivity`. This superclass is the Mobile SDK's basic abstract activity class. `SalesforceActivity`, 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 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:

```
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:

```
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) {
            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(Exception exception)
        {
            Toast.makeText(MainActivity.this,
                MainActivity.this.getString(
                    SalesforceSDKManager.getInstance().getSalesforceR().stringGenericError(),
                    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.

Java Note:

In the call to `RestClient.sendAsync()` the code instantiates a new `AsyncRequestCallback` object as its second argument. However, the `AsyncRequestCallback` constructor 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` an `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.

TemplateApp Manifest

A look at the `AndroidManifest.xml` file in the `TemplateApp` project reveals the components required for Mobile SDK native Android apps. The only required component is:

Name	Type	Description
<code>MainActivity</code>	Activity	The first activity to be called after login. The name and the class are defined in the project.

Because any app created by the `create_native` script is based on the `TemplateApp` 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, using the Android Manifest editor in Eclipse.

Tutorial: Creating a Native Android Warehouse Application

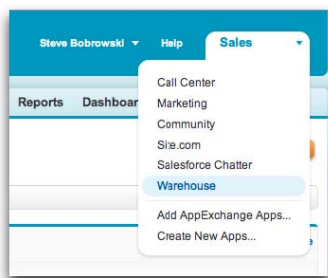
Apply your knowledge of the native Android SDK by building a mobile inventory management app. This tutorial demonstrates a simple master-detail architecture that defines two activities. It demonstrates Mobile SDK application setup, use of REST API wrapper classes, and Android SDK integration.

Prerequisites

This tutorial requires the following tools and packages.

- This tutorial uses a Warehouse app that contains a basic inventory database. You'll need to install this app in a DE org. If you install it in an existing DE org, be sure to delete any existing Warehouse components you've made before you install.
 - Click the installation URL link: <https://login.salesforce.com/package/installPackage.apexp?p0=04ti0000000MMMT>

2. If you aren't logged in already, enter the username and password of your DE org.
3. On the Package Installation Details page, click **Continue**.
4. Click **Next**, and on the Security Level page click **Next**.
5. Click **Install**.
6. Click **Deploy Now** and then **Deploy**.
7. Once the installation completes, you can select the **Warehouse** app from the app picker in the upper right corner.



8. To create data, click the **Data** tab.
9. Click the **Create Data** button.

- Install the latest versions of:

- ◇ Java JDK 6 or higher—<http://www.oracle.com/downloads>.
- ◇ Apache Ant 1.8 or later—<http://ant.apache.org>.
- ◇ Android SDK Tools, version 21 or later—<http://developer.android.com/sdk/installing.html>.



Note: For best results, install all previous versions of the Android SDK as well as your target version.

- ◇ Eclipse—<https://www.eclipse.org>. Check the [Android Development Tools website](#) for the minimum supported Eclipse version.
- ◇ Android ADT (Android Development Tools) plugin for Eclipse, version 21 or later—<http://developer.android.com/sdk>.
- ◇ In order to run the application in the Emulator, you need to set up at least one Android Virtual Device (AVD) that targets Platform 2.2 or above (we recommend 4.0 or above). To learn how to set up an AVD in Eclipse, follow the instructions at <http://developer.android.com/guide/developing/devices/managing-avds.html>.
- Install the Salesforce Mobile SDK using npm:
 1. If you've already successfully installed Node.js and npm, skip to step 4.
 2. Install Node.js on your system. The Node.js installer automatically installs npm.
 - i. Download Node.js from www.nodejs.org/download.
 - ii. Run the downloaded installer to install Node.js and npm. Accept all prompts asking for permission to install.
 3. At the Terminal window, type `npm` and press `Return` to make sure your installation was successful. If you don't see a page of usage information, revisit Step 2 to find out what's missing.
 4. At the Terminal window, type `sudo npm install forcedroid -g`

This command uses the forcedroid package to install the Mobile SDK globally. With the `-g` option, you can run `npm install` from any directory. The npm utility installs the package under `/usr/local/lib/node_modules`, and links binary modules in `/usr/local/bin`. Most users need the `sudo` option because they lack read-write permissions in `/usr/local`.

Create a Native Android App

In this tutorial, you learn how to get started with the Salesforce Mobile SDK, including how to install the SDK and a quick tour of the native project template using your DE org. Subsequent tutorials show you how to modify the template app and make it work with the Warehouse schema.

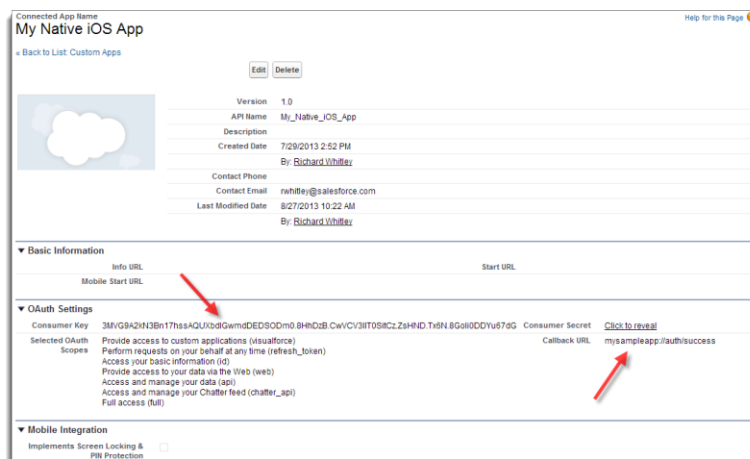
Step 1: Create a Connected App

In this step, you learn how to configure a Connected App in Force.com. Doing so authorizes the mobile app you will soon build to communicate securely with Force.com and access Force.com APIs on behalf of users via the industry-standard OAuth 2.0 protocol.

1. In your DE org, click **Your Name** > **Setup** then click **Create** > **Apps**.
2. Under **Connected Apps**, click **New** to bring up the **New Connected App** page.
3. Under **Basic Information**, fill out the form as follows:
 - **Connected App Name:** My Native Android App
 - **API Name:** accept the suggested value
 - **Contact Email:** enter your email address
4. Under OAuth Settings, check the **Enable OAuth Settings** checkbox.
5. Set **Callback URL** to `mysampleapp://auth/success`.
6. Under **Available OAuth Scopes**, check “Access and manage your data (api)” and “Perform requests on your behalf at any time (refresh_token)”, then click **Add**.
7. Click **Save**.

After you save the configuration, notice the details of the Connected App you just created.

- Note the Callback URL and Consumer Key; you will use these when you set up your native app in the next step.
- Mobile apps do not use the Consumer Secret, so you can ignore this value.



Step 2: Create a Native Android Project

To create a new Mobile SDK project, use the forcedroid utility again in the Terminal window.

1. Change to the directory in which you want to create your project.

2. To create an Android project, type `forcedroid create`.

The `forcedroid` utility prompts you for each configuration value.

3. For application type, enter `native`.
4. For application name, enter `Warehouse`.
5. For target directory, enter `tutorial/AndroidNative`.
6. For package name, enter `com.samples.warehouse`.
7. When asked if you want to use SmartStore, press **Return** to accept the default.

Step 3: Run the New Android App

Now that you've successfully created a new Android app, build and run it in Eclipse to make sure that your environment is properly configured.



Note: If you run into problems, first check the Android SDK Manager to make sure that you've got the latest Android SDK, build tools, and development tools. You can find the Android SDK Manager under **Window > Android SDK Manager** in Eclipse. After you've installed anything that's missing, close and restart Android SDK Manager to make sure you're up-to-date.

Importing and Building Your App in Eclipse

The `forcedroid` script prints instructions for running the new app in the Eclipse editor.

1. Launch Eclipse and select `tutorial/AndroidNative` as your workspace directory.
2. Select **Eclipse > Preferences**, choose the **Android** section, and enter the Android SDK location.
3. Click **OK**.
4. Select **File > Import** and select **General > Existing Projects into Workspace**.
5. Click **Next**.
6. Specify the `forcedroid/native` directory as your root directory. Next to the list that displays, click **Deselect All**, then browse the list and check the `SalesforceSDK` project.
7. Click **Finish**.
8. Repeat Steps 4–8. In Step 6, choose `tutorial/AndroidNative` as the root, then select only your new `Warehouse` project.

When you've finished importing the projects, Eclipse automatically builds your workspace. This process can take several minutes. When the status bar reports zero errors, you're ready to run the project.

1. In your Eclipse workspace, Control-click or right-click your project.
2. From the popup menu, choose **Run As > Android Application**.



Note: If the **Run As** menu doesn't include **Android Application**, you need to configure an Android emulator or device.

Eclipse launches your app in the emulator or on your connected Android device.

Step 4: Explore How the Android App Works

The native Android app uses a straightforward Model View Controller (MVC) architecture.

- The model is the Warehouse database schema
- The views come from the activities defined in your project
- The controller functionality represents a joint effort between the Android SDK classes, the Salesforce Mobile SDK, and your app

Within the view, the finished tutorial app defines two Android activities in a master-detail relationship. `MainActivity` lists records from the Merchandise custom objects. `DetailActivity`, which you access by clicking on an item in `MainActivity`, lets you view and edit the fields in the selected record.

MainActivity Class

When the app is launched, the `WarehouseApp` class initially controls the execution flow. After the login process completes, the `WarehouseApp` instance passes control to the main activity class, via the `SalesforceSDKManager` singleton.

In the template app that serves as the basis for your new app, and also in the finished tutorial, the main activity class is named `MainActivity`. This class subclasses `SalesforceActivity`, which is the Mobile SDK base class for all activities.

Before it's customized, though, the app doesn't include other activities or touch event handlers. It simply logs into Salesforce, issues a request using Salesforce Mobile SDK REST APIs, and displays the response in the main activity. In this tutorial you replace the template app controls and repurpose the SOQL REST request to work with the Merchandise custom object from the Warehouse schema.

DetailActivity Class

The `DetailActivity` class also subclasses `SalesforceActivity`, but it demonstrates more interesting customizations. `DetailActivity` implements text editing using standard Android SDK classes and XML templates. It also demonstrates how to update a database object in Salesforce using the `RestClient` and `RestRequest` classes from the Mobile SDK.

RestClient and RestRequest Classes

Mobile SDK apps interact with Salesforce data through REST APIs. However, you don't have to construct your own REST requests or work directly at the HTTP level. You can process SOQL queries, do SOSL searches, and perform CRUD operations with minimal coding by using static convenience methods on the `RestRequest` class. Each `RestRequest` convenience method returns a `RestRequest` object that wraps the formatted REST request.

To send the request to the server, you simply pass the `RestRequest` object to the `sendAsync()` or `sendSync()` method on your `RestClient` instance. You don't create `RestClient` objects. If your activity inherits a Mobile SDK activity class such as `SalesforceActivity`, Mobile SDK passes an instance of `RestClient` to the `onResume()` method. Otherwise, you can call `ClientManager.getRestClient()`. Your app uses the connected app information from your `bootconfig.xml` file so that the `RestClient` object can send REST requests on your behalf.

Customize the List Screen

In this tutorial, you modify the main activity and its layout to make the app specific to the Warehouse schema. You also adapt the existing SOQL query to obtain all the information we need from the Merchandise custom object.

Step 1: Remove Existing Controls

The template code provides a main activity screen that doesn't suit our purposes. Let's gut it to make room for our code.

1. From the Package Explorer in Eclipse, open the `res/layout/main.xml` file. Make sure to set the view to text mode. This XML file contains a `<LinearLayout>` root node, which contains three child nodes: an `<include>` node, a nested `<LinearLayout>` node, and a `<ListView>` node.

2. Delete the nested `<LinearLayout>` node that contains the three `<Button>` nodes. The edited file looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent" android:background="#454545"
    android:id="@+id/root">

    <include layout="@layout/header" />

    <ListView android:id="@+id/contacts_list" android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
</LinearLayout>
```

3. Save the file, then open the `src/com.samples.warehouse/MainActivity.java` file.
4. Delete the `onClearClick()`, `onFetchAccountsClick()`, and `onFetchContactsClick()` methods. If the compiler warns you that the `sendRequest()` method is never used locally, that's OK. You just deleted all calls to that method, but you'll fix that in the next step.

Step 2: Update the SOQL Query

The `sendRequest()` method provides code for sending a SOQL query as a REST request. You can reuse some of this code while customizing the rest to suit your new app.

1. Rename `sendRequest()` to `fetchDataForList()`. Replace

```
private void sendRequest(String soql) throws UnsupportedOperationException
```

with

```
private void fetchDataForList()
```

Note that you've removed the `throw` declaration. You'll reinstate it within the method body to keep the exception handling local. You'll add a `try...catch` block around the call to `RestRequest.getRequestForQuery()`, rather than throwing exceptions to the `fetchDataForList()` caller.

2. Add a hard-coded SOQL query that returns up to 10 records from the `Merchandise__c` custom object:

```
private void fetchDataForList() {
    String soql = "SELECT Name, Id, Price__c, Quantity__c
        FROM Merchandise__c LIMIT 10";
```

3. Wrap a `try...catch` block around the call to `RestRequest.getRequestForQuery()`. Replace this:

```
RestRequest restRequest = RestRequest.getRequestForQuery(getString(R.string.api_version),
    soql);
```

with this:

```
RestRequest restRequest = null;
try {
    restRequest =
        RestRequest.getRequestForQuery(getString(R.string.api_version), soql);
} catch (UnsupportedEncodingException e) {
```

```

        showError(MainActivity.this, e);
        return;
    }

```

Here's the completed version of what was formerly the `sendRequest()` method:

```

private void fetchDataForList() {
    String soql = "SELECT Name, Id, Price__c, Quantity__c FROM
        Merchandise__c LIMIT 10";
    RestRequest restRequest = null;
    try {
        restRequest =
            RestRequest.getRequestForQuery(
                getString(R.string.api_version), soql);
    } catch (UnsupportedEncodingException e) {
        showError(MainActivity.this, e);
        return;
    }

    client.sendAsync(restRequest, new AsyncRequestCallback() {
        @Override
        public void onSuccess(RestRequest request,
            RestResponse result) {
            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(Exception exception) {
            Toast.makeText(MainActivity.this,
                MainActivity.this.getString(
                    SalesforceSDKManager.getInstance().
                        getSalesforceR().stringGenericError(),
                    exception.toString()),
                Toast.LENGTH_LONG).show();
        }
    });
}

```

We'll call `fetchDataForList()` when the screen loads, after authentication completes.

4. In the `onResume(RestClient client)` method, add the following line at the end of the method body:

```

@Override
public void onResume(RestClient client) {
    // Keeping reference to rest client
    this.client = client;

    // Show everything
    findViewById(R.id.root).setVisibility(View.VISIBLE);
    // Fetch data for list
    fetchDataForList();
}

```

5. Finally, implement the `showError()` method to report errors through a given activity context. At the top of the file, add the following line to the end of the list of imports:

```
import android.content.Context;
```

6. At the end of the `MainActivity` class definition add the following code:

```
public static void showError(Context context, Exception e) {
    Toast toast = Toast.makeText(context,
        context.getString(
            SalesforceSDKManager.getInstance().getSalesforceR().stringGenericError(),
            e.toString()),
        Toast.LENGTH_LONG);
    toast.show();
}
```

7. Save the `MainActivity.java` file.

Step 3: Try Out the App

To test the app, Control-Click the app in Package Explorer and select **Run As > Android Application**. When the Android emulator displays, wait a few minutes as it loads. Unlock the screen and wait a while longer for the Salesforce login screen to appear. After you log into Salesforce successfully, click **Allow** to give the app the permissions it requires.

At this point, if you click a Merchandise record, nothing happens. You'll fix that in the next tutorial.

Create the Detail Screen

In the previous step, you modified the template app so that the main activity presents a list of up to ten Merchandise records. In this step, you finish the job by creating a detail activity and layout. You then link the main activity and the detail activity.

Step 1: Create the Detail Screen

To start, design the layout of the detail activity by creating an XML file named `res/layout/detail.xml`.

1. In Package Explorer, expand `res/layout`.
2. Control-click the layout folder and select **New > Android XML File**.
3. In the **File** field, type `detail.xml`.
4. Under **Root Element**, select **LinearLayout**.
5. Click **Finish**.

In the new file, define layouts and resources to be used in the detail screen. Start by adding fields and labels for name, price, and quantity.

6. Replace the contents of the new file with the following XML code.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/root"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#454545"
    android:orientation="vertical" >
```

```

<include layout="@layout/header" />

<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/name_label"
        android:width="100dp" />

    <EditText
        android:id="@+id/name_field"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:inputType="text" />
</LinearLayout>

<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/price_label"
        android:width="100dp" />

    <EditText
        android:id="@+id/price_field"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:inputType="numberDecimal" />
</LinearLayout>

<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/quantity_label"
        android:width="100dp" />

    <EditText
        android:id="@+id/quantity_field"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:inputType="number" />
</LinearLayout>
</LinearLayout>

```

7. Save the file.
8. To finish the layout, define the display names for the three labels (`name_label`, `price_label`, and `quantity_label`) referenced in the `TextView` elements.

Add the following to `res/values/strings.xml` just before the close of the `<resources>` node:

```
<!-- Detail screen -->
<string name="name_label">Name</string>
<string name="price_label">Price</string>
<string name="quantity_label">Quantity</string>
```

9. Save the file, then open the `AndroidManifest.xml` file in text view. If you don't get the text view, click the **AndroidManifest.xml** tab at the bottom of the editor screen.
10. Declare the new activity in `AndroidManifest.xml` by adding the following in the `<application>` section:

```
<!-- Merchandise detail screen -->
<activity android:name="com.samples.warehouse.DetailActivity"
    android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
</activity>
```

Except for a button that we'll add later, you've finished designing the layout and the string resources for the detail screen. To implement the screen's behavior, you define a new activity.

Step 2: Create the DetailActivity Class

In this module we'll create a new class file named `DetailActivity.java` in the `com.samples.warehouse` package.

1. In Package Explorer, expand the **WarehouseApp > src > com.samples.warehouse** node.
2. Control-click the `com.samples.warehouse` folder and select **New > Class**.
3. In the **Name** field, enter **DetailActivity**.
4. In the **Superclass** field, enter or browse for `com.salesforce.androidsdk.ui.sfnative.SalesforceActivity`.
5. Click **Finish**.

The compiler provides a stub implementation of the required `onResume()` method. Mobile SDK passes an instance of `RestClient` to this method. Since you need this instance to create REST API requests, it's a good idea to cache a reference to it.

6. Add the following declaration to the list of member variables at the top of the new class:

```
private RestClient client;
```

7. In the `onResume()` method body, add the following code:

```
@Override
public void onResume(RestClient client) {
    // Keeping reference to rest client
    this.client = client;
}
```

Step 3: Customize the DetailActivity Class

To complete the activity setup, customize the `DetailActivity` class to handle editing of Merchandise field values.

1. Add the following imports to the list of imports at the top of `DetailActivity.java`:

```
import android.widget.EditText;
import android.os.Bundle;
```

2. At the top of the class body, add private `EditText` members for the three input fields.

```
private EditText nameField;
private EditText priceField;
private EditText quantityField;
```

3. Add a variable to contain a record ID from the `Merchandise` custom object. You'll add code to populate it later when you link the main activity and the detail activity.

```
private String merchandiseId;
```

4. Add an `onCreate()` method that configures the view to use the `detail.xml` layout you just created. Place this method just before the end of the class definition.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Setup view
    setContentView(R.layout.detail);
    nameField = (EditText) findViewById(R.id.name_field);
    priceField = (EditText) findViewById(R.id.price_field);
    quantityField = (EditText)
        findViewById(R.id.quantity_field);
}
```

Step 4: Link the Two Activities, Part 1: Create a Data Class

Next, you need to hook up `MainActivity` and `DetailActivity` classes so they can share the fields of a selected `Merchandise` record. When the user clicks an item in the inventory list, `MainActivity` needs to launch `DetailActivity` with the data it needs to display the record's fields.

Right now, the list adapter in `MainActivity.java` is given only the names of the `Merchandise` fields. Let's store the values of the standard fields (`id` and `name`) and the custom fields (`quantity`, and `price`) locally so you can send them to the detail screen.

To start, define a static data class to represent a `Merchandise` record.

1. In the Package Explorer, open **src > com.samples.warehouse > MainActivity.java**.
2. Add the following class definition at the end of the `MainActivity` definition:

```
/**
 * Simple class to represent a Merchandise record
 */
static class Merchandise {
    public final String name;
    public final String id;
    public final int quantity;
    public final double price;
```



```

public Merchandise(String name, String id, int quantity, double price) {
    this.name = name;
    this.id = id;
    this.quantity = quantity;
    this.price = price;
}

public String toString() {
    return name;
}
}

```

3. To put this class to work, modify the main activity's list adapter to take a list of Merchandise objects instead of strings. In the `listAdapter` variable declaration, change the template type from `String` to `Merchandise`:

```
private ArrayAdapter<Merchandise> listAdapter;
```

4. To match the new type, change the `listAdapter` instantiation in the `onResume()` method:

```
listAdapter = new ArrayAdapter<Merchandise>(this, android.R.layout.simple_list_item_1,
    new ArrayList<Merchandise>());
```

Next, modify the code that populates the `listAdapter` object when the response for the `SOQL` call is received.

5. Add the following import to the existing list at the top of the file:

```
import org.json.JSONObject;
```

6. Change the `onSuccess()` method in `fetchDataForList()` to use the new `Merchandise` object:

```

public void onSuccess(RestRequest request, RestResponse result) {
    try {
        listAdapter.clear();
        JSONArray records = result.asJSONObject().getJSONArray("records");
        for (int i = 0; i < records.length(); i++) {
            JSONObject record = records.getJSONObject(i);
            Merchandise merchandise = new Merchandise(record.getString("Name"),
                record.getString("Id"), record.getInt("Quantity__c"),
                record.getDouble("Price__c"));
            listAdapter.add(merchandise);
        }
    } catch (Exception e) {
        onError(e);
    }
}

```

Step 5: Link the Two Activities, Part 2: Implement a List Item Click Handler

Next, you need to catch click events and launch the detail screen when these events occur. Let's make `MainActivity` the listener for clicks on list view items.

1. Open the `MainActivity.java` file in the editor.

2. Add the following import:

```
import android.widget.AdapterView.OnItemClickListener;
```

3. Change the class declaration to implement the `OnItemClickListener` interface:

```
public class MainActivity extends SalesforceActivity implements OnItemClickListener {
```

4. Add a private member for the list view:

```
private ListView listView;
```

5. Add the following code in bold to the `onResume()` method just before the `super.onResume()` call:

```
public void onResume() {
    // Hide everything until we are logged in
    findViewById(R.id.root).setVisibility(View.INVISIBLE);

    // Create list adapter
    listAdapter = new ArrayAdapter<Merchandise>(
        this, android.R.layout.simple_list_item_1, new ArrayList<Merchandise>());
    ((ListView) findViewById(R.id.contacts_list)).setAdapter(listAdapter);

    // Get a handle for the list view
    listView = (ListView) findViewById(R.id.contacts_list);
    listView.setOnItemClickListener(this);

    super.onResume();
}
```

Now that you've designated a listener for list item clicks, you're ready to add the list item click handler.

6. Add the following imports:

```
import android.widget.AdapterView;
import android.content.Intent;
```

7. Just before the `Merchandise` class definition, add an `onItemClick()` method.

```
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
}
```

8. Get the selected item from the list adapter in the form of a `Merchandise` object.

```
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
    Merchandise merchandise = listAdapter.getItem(position);
}
```

9. Create an Android intent to start the detail activity, passing the merchandise details into it.

```
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
    Merchandise merchandise = listAdapter.getItem(position);
    Intent intent = new Intent(this, DetailActivity.class);
}
```

```

intent.putExtra("id", merchandise.id);
intent.putExtra("name", merchandise.name);
intent.putExtra("quantity", merchandise.quantity);
intent.putExtra("price", merchandise.price);
startActivity(intent);
}

```

Let's finish by updating the `DetailActivity` class to extract the merchandise details from the intent.

10. In the Package Explorer, open **src > com.samples.warehouse > DetailActivity.java**.
11. In the `onCreate()` method, assign values from the list screen selection to their corresponding data members in the detail activity:

```

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Setup view
    setContentView(R.layout.detail);
    nameField = (EditText) findViewById(R.id.name_field);
    priceField = (EditText) findViewById(R.id.price_field);
    quantityField = (EditText)
        findViewById(R.id.quantity_field);
    // Populate fields with data from intent
    Bundle extras = getIntent().getExtras();
    merchandiseId = extras.getString("id");
    nameField.setText(extras.getString("name"));
    priceField.setText(extras.getDouble("price") + "");
    quantityField.setText(extras.getInt("quantity") + "");
}

```

Step 6: Implement the Update Button

You're almost there! The only part of the UI that's missing is a button that writes the user's edits to the server. You need to:

- Add the button to the layout
- Define the button's label
- Implement a click handler
- Implement functionality that saves the edits to the server

1. Reopen `detail.xml` and add the following `<Button>` node as the last node in the outermost layout.

```

<Button
    android:id="@+id/update_button"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:onClick="onUpdateClick"
    android:text="@string/update_button" />

```

2. Save the `detail.xml` file, then open `strings.xml`.
3. Add the following button label string to the end of the list of strings:

```

<string name="update_button">Update</string>

```

4. Save the `strings.xml` file, then open `DetailActivity.java`.

In the `DetailActivity` class, add a handler for the Update button's `onClick` event. The handler's name must match the `android:onClick` value in the `<Button>` node that you just added to `detail.xml`. In this case, the name is `onUpdateClick`. This method simply creates a map that matches `Merchandise__c` field names to corresponding values in the detail screen. Once the values are set, it calls the `saveData()` method to write the changes to the server.

5. To support the handler, add the following imports to the existing list at the top of the file:

```
import java.util.HashMap;
import java.util.Map;
import android.view.View;
```

6. Add the following method to the `DetailActivity` class definition:

```
public void onUpdateClick(View v) {
    Map<String, Object> fields = new HashMap<String, Object>();
    fields.put("Name", nameField.getText().toString());
    fields.put("Quantity__c", quantityField.getText().toString());
    fields.put("Price__c", priceField.getText().toString());
    saveData(merchandiseId, fields);
}
```

The compiler reminds you that `saveData()` isn't defined. Let's fix that. The `saveData()` method creates a REST API update request to update the `Merchandise__c` object with the user's values. It then sends the request asynchronously to the server using the `RestClient.sendAsync()` method. The callback methods that receive the server response (or server error) are defined inline in the `sendAsync()` call.

7. Add the following imports to the existing list at the top of the file:

```
import com.salesforce.androidsdk.rest.RestRequest;
import com.salesforce.androidsdk.rest.RestResponse;
```

8. Implement the `saveData()` method in the `DetailActivity` class definition:

```
private void saveData(String id, Map<String, Object> fields) {
    RestRequest restRequest;
    try {
        restRequest = RestRequest.getRequestForUpdate(getString(R.string.api_version),
            "Merchandise__c", id, fields);
    } catch (Exception e) {
        // You might want to log the error or show it to the user
        return;
    }

    client.sendAsync(restRequest, new RestClient.AsyncRequestCallback() {
        @Override
        public void onSuccess(RestRequest request, RestResponse result) {
            try {
                DetailActivity.this.finish();
            } catch (Exception e) {
                // You might want to log the error or show it to the user
            }
        }

        @Override
        public void onError(Exception e) {
            // You might want to log the error or show it to the user
        }
    });
}
```

That's it! Your app is ready to run and test.

Step 7: Try Out the App

1. Build your app and run it in the Android emulator. If you did everything correctly, a detail page appears when you click a Merchandise record in the Warehouse screen.
2. Update a record's quantity and price. Be sure to click the **Update** button in the detail view after you edit the values. When you navigate back to the detail view, the updated values display.
3. Log into your DE org and view the record using the browser UI to see the updated values.

Android Native Sample Applications

Salesforce Mobile SDK includes the following native Android sample applications.

- **RestExplorer** demonstrates the OAuth and REST API functions of the SalesforceSDK. It's also useful for investigating REST API actions from a Honeycomb tablet.
 1. To run the application from your Eclipse workspace, right-click the **RestExplorer** project and choose **Run As > Android Application**.
 2. To run the tests, right-click the **RestExplorerTest** project and choose **Run As > Android JUnit Test**.
- **NativeSqlAggregator** demonstrates SQL aggregation with SmartSQL. As such, it also demonstrates a native implementation of SmartStore. To run the application from your Eclipse workspace, right-click the **NativeSqlAggregator** project and choose **Run As > Android Application**.
- **FileExplorer** demonstrates the Files API as well as the underlying Google Volley networking enhancements. To run the application from your Eclipse workspace, right-click the **FileExplorer** project and choose **Run As > Android Application**.

Chapter 6

Files and Networking

In this chapter ...

- [Architecture](#)
- [Downloading Files and Managing Sharing](#)
- [Uploading Files](#)
- [Encryption and Caching](#)
- [Using Files in Android Apps](#)
- [Using Files in iOS Native Apps](#)
- [Using Files in Hybrid Apps](#)

Mobile SDK 2.1 introduces an API for files and networking. This API includes two levels of technology. For file management, the SDK provides a set of convenience methods that wraps the file requests in the Chatter REST API. Under the REST API wrapper level, a networking layer exposes objects that let the app control pending REST requests. Together, these two sides of the same coin give the SDK a more robust feature set as well as enhanced networking performance.

Architecture

Beginning with Mobile SDK 2.1, the Android REST request system uses Google Volley, an open-source external library, as its underlying architecture. This architecture allows you to access the Volley `QueueManager` object to manage requests. At runtime you can use the `QueueManager` to cancel pending requests on asynchronous threads. You can learn about Volley at <https://developers.google.com/events/io/sessions/325304728>

In iOS, file management and networking rely on the `SalesforceNetworkSDK` library. All REST API call—for files as well as any other REST requests—go through this library. The `SalesforceNetworkSDK` library itself implements `MKNetworkKit`, a popular open source library.



Note: If you directly accessed the `RestKit` library in old versions of your Mobile SDK iOS app, you'll need to update that code to use the `MKNetworkKit` library.

Hybrid JavaScript functions use the architecture of the Mobile SDK for the device operating system (Android or iOS) to implement file operations. These functions are defined in `forcetk.mobilesdk.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	<code>FileRequests.uploadFile()</code>	<pre>public static RestRequest uploadFile(File theFile, String name, String description, String mimeType) throws UnsupportedOperationException</pre>
iOS native	<pre>- requestForUploadFile: name:description:mimeType:</pre>	<pre>- (SFRestRequest *) requestForUploadFile:(NSData *)data name:(NSString *)name description:(NSString *)description mimeType:(NSString *)mimeType</pre>
Hybrid (Android and iOS)	N/A	N/A

Encryption and Caching

Mobile SDK 2.1 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.

Managing the Request Queue

The `RestClient` class internally uses an instance of the Volley `RequestQueue` class to manage REST API requests. You can access the underlying `RequestQueue` object by calling `restClient.getRequestQueue()` on your `RestClient` instance. With the `RequestQueue` object you can directly cancel and otherwise manipulate pending requests.

Example: Canceling All Pending Requests

The following code calls `getRequestQueue()` on an instance of `RestClient` (`client`). It then calls the `RequestQueue.cancelAll()` method to cancel all pending requests in the queue. The `cancelAll()` method accepts

a `RequestFilter` parameter, so the code passes in an object of a custom class, `CountingFilter`, which implements the `Volley RequestFilter` interface.

```
CountingFilter countingFilter = new CountingFilter();
client.getRequestQueue().cancelAll(countingFilter);
int count = countingFilter.getCancelCount();
...

/**
 * Request filter that cancels all requests and also counts the number of requests canceled
 *
 */
class CountingFilter implements RequestFilter {

    private int count = 0;

    public int getCancelCount() {
        return count;
    }

    @Override
    public boolean apply(Request<?> request) {
        count++;
        return true;
    }
}
```

`RequestQueue.cancelAll()` lets the `RequestFilter`-based object inspect each item in the queue before allowing the operation to continue. Internally, `cancelAll()` calls the filter's `apply()` method on each iteration. If `apply()` returns true, the cancel operation continues. If it returns false, `cancelAll()` does not cancel that request and continues to the next request in the queue.

In this code example, the `CountingFilter.apply()` merely increments an internal counter on each call. After the `cancelAll()` operation finishes, the sample code calls `CountingFilter.getCancelCount()` to report the number of canceled objects.

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.

iOS Project Settings

If you're updating Salesforce Mobile SDK apps built prior to Mobile SDK 2.1, you'll need to adjust your project settings for all targets to include the `SalesforceNetworkSDK` library.

1. Download the `MKNetworking` library bundled with Mobile SDK 2.1. Get the binary libraries and their headers from compressed files at <https://github.com/forcedotcom/SalesforceMobileSDK-iOS-Distribution>.
2. Link the following modules in your project:
 - `libMKNetworkKit-iOS.a`
 - `libSalesforceNetworkSDK.a`
 - `ImageIO.framework`

The Files API also requires the following frameworks which are normally linked by default:

- `CFNetwork.framework`
- `SystemConfiguration.framework`
- `Security.framework`

REST Responses and Multithreading

The networking 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.

Managing Requests

MKNetworkKit, the underlying networking architecture for the iOS Mobile SDK, uses two key objects: `MKNetworkEngine` and `MKNetworkOperation`. The Salesforce Network SDK for iOS in turn defines two primary objects, `SFNetworkEngine` and `SFNetworkOperation`, that wrap the corresponding MKNetworkKit objects. `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;
```

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

Except for uploading, you can use the same file requests in hybrid apps as in native apps. Hybrid file request wrappers reside in the `forcetk.mobilesdk.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 leverage the `SmartSync.js` and `forcetk.mobilesdk.js` libraries to build your HTML app. The `HybridFileExplorer` sample app demonstrates this.



Note: Mobile SDK does not support file uploads in hybrid apps.

Chapter 7

Offline Management

In this chapter ...

- [Securely Storing Data Offline](#)
- [Using SmartSync to Access Salesforce Objects](#)

Salesforce Mobile SDK provides two modules that help you store and synchronize data for offline use:

- SmartStore
- SmartSync Data Framework

SmartStore lets you store app data in encrypted databases, or *soups*, on the device. When the device goes back online, you can use SmartStore APIs to synchronize data changes with the Salesforce server.

SmartSync provides a mechanism for easily fetching Salesforce data, modeling it as JavaScript objects, and caching it for offline use. When it's time to upload offline changes to the Salesforce server, SmartSync gives you highly granular control over the synchronization process. SmartSync is built on the popular `Backbone.js` open source library and uses SmartStore as its default cache.

Securely Storing Data Offline

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 uses SmartStore, a multi-threaded, secure offline storage solution for mobile devices. SmartStore allows you to continue working with data even when the device is not connected to the Internet.

About SmartStore

SmartStore stores data as JSON documents in a simple, single-table database. You can define indexes for this database, and you can query the data either with SmartStore helper methods that implement standard queries, or with custom queries using SmartStore's Smart SQL language.

SmartStore uses StoreCache, a Mobile SDK caching mechanism, to provide offline synchronization and conflict resolution services. 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 the 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.

Sample Objects

The code snippets in this chapter use two objects, Account and Opportunity, which come predefined with every Salesforce organization. Account and Opportunity have a master-detail relationship; an account can have more than one opportunity.

SmartStore Soups

SmartStore stores offline data in one or more *soups*. A soup, conceptually speaking, is a logical collection of data records—represented as JSON objects—that you want to store and query offline. In the Force.com world, a soup typically maps to a standard or custom object that you wish to store offline. You can store as many soups as you want in an application, but remember that soups are meant to be self-contained data sets; there is no direct correlation between soups. In addition to storing the data, you can also specify indices that map to fields within the data, for greater ease in customizing data queries.



Warning:

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 organization sets a short lifetime for the refresh token.

SmartStore Data Types

SmartStore supports the following data types:

Type	Description
integer	Signed integer, stored in 4 bytes (SDK 2.1 and earlier) or 8 bytes (SDK 2.2 and later)

Type	Description
floating	Floating point value, stored as an 8-byte IEEE floating point number
string	Text string, stored with database encoding (UTF-8)

Date Representation

Date Representation

SmartStore does not specify a type for dates and times. We recommend that you represent these values with SmartStore data types as shown in 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 Apps

Hybrid apps access SmartStore from JavaScript. To enable offline access in a hybrid mobile application, your Visualforce or HTML page must include the following JavaScript library files.

- `cordova-x.x.x.js`—The Cordova library (formerly PhoneGap).
- `cordova.force.js`—Contains the JavaScript portion of Salesforce OAuth and SmartStore plug-ins. Also includes methods that perform utility tasks such as determining whether you're offline.

In Android apps, SmartStore is an optional component. It is not optional in iOS apps. When you use the Mobile SDK npm scripts to create SmartStore apps:

- If you create an iOS hybrid project by using `forceios create`, these libraries are automatically included.
- If you create an Android hybrid project by using `forcedroid create` with prompts, specify `yes` when you're asked if you want to use SmartStore.
- If you're using `forcedroid create` with command-line parameters, specify the optional `--usesmartstore=true` parameter.

Adding SmartStore to Existing Android Apps

To add SmartStore to an existing Android project (hybrid or native):

1. Add the SmartStore library project to your project. In Eclipse, choose Properties from the Project menu. Select Android from the left panel, then click **Add** on the right-hand side. Choose the `hybrid/SmartStore` project.

2. In your `<projectname>App.java` file, import the `SalesforceSDKManagerWithSmartStore` class instead of `SalesforceSDKManager`. Replace this statement:

```
import com.salesforce.androidsdk.app.SalesforceSDKManager
```

with this one:

```
import com.salesforce.androidsdk.smartstore.app.SalesforceSDKManagerWithSmartStore
```

3. In your `<projectname>App.java` file, change your `App` class to extend the `SalesforceSDKManagerWithSmartStore` class rather than `SalesforceSDKManager`.

Registering a Soup

In order to access a soup, you first need to register it. Provide a name, index specifications, and names of callback functions for success and error conditions:

```
navigator.smartstore.registerSoup(soupName, indexSpecs, successCallback, errorCallback)
```

If the soup does not already exist, this function creates it. If the soup already exists, registering gives you access to the existing soup. To find out if a soup already exists, use:

```
navigator.smartstore.soupExists(soupName, successCallback, errorCallback);
```

A soup is indexed on one or more fields found in its entries. Insert, update, and delete operations on soup entries are tracked in the soup indices. 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.

indexSpecs

The `indexSpecs` array is used to create the soup with predefined indexing. Entries in the `indexSpecs` array specify how the soup should be indexed. Each entry consists of a `path:type` pair. `path` is the name of an index field; `type` is either “string”, “integer”, or “floating”. Index paths are case-sensitive and can include compound paths, such as `Owner.Name`.



Note: If index entries are missing any fields described in a particular `indexSpec`, they will not be tracked in that index.

```
"indexSpecs": [
  {
    "path": "Name",
    "type": "string"
  },
  {
    "path": "Id",
    "type": "string"
  },
  {
    "path": "ParentId",
    "type": "string"
  },
  {
    "path": "lastModifiedDate",
    "type": "integer"
  }
]
```

```
}
]
```

**Note:**

- 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 the type that you specified in the index specification.
- It’s OK to have a null field in an index column.

successCallback

The success callback function for `registerSoup` takes one argument (the soup name).

```
function(soupName) { alert("Soup " + soupName + " was successfully created"); };
```

A successful creation of the soup returns a `successCallback` that indicates 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 for `registerSoup` takes one argument (the error description string).

```
function(err) { alert ("registerSoup failed with error:" + err); }
```

During soup creation, errors can happen for a number of 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

Retrieving Data From a Soup

SmartStore provides a set of helper methods that build query strings for you. To query a specific set of records, call the `build*` method that suits your query specification. 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
<code>indexPath</code>	This is what you’re searching for; for example a name, account number, or date.
<code>beginKey</code>	Optional. Used to define the start of a range query.
<code>endKey</code>	Optional. Used to define the end of a range query.
<code>order</code>	Optional. Either “ascending” or “descending.”
<code>pageSize</code>	Optional. If not present, the native plugin can return whatever page size it sees fit in the resulting <code>Cursor.pageSize</code> .

**Note:**

All queries are single-predicate searches. Only Smart SQL queries support joins.

Query Everything

`buildAllQuerySpec(indexPath, order, [pageSize])` returns all entries in the soup, with no particular order. Use this query to traverse everything in the soup.

`order` and `pageSize` are optional, and default to ascending and 10, respectively. You can specify:

- `buildAllQuerySpec(indexPath)`
- `buildAllQuerySpec(indexPath, order)`
- `buildAllQuerySpec(indexPath, order, [pageSize])`

However, you can't specify `buildAllQuerySpec(indexPath, [pageSize])`.

See [Working With Cursors](#) 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 might want to increase the value to two or three times the number of entries actually shown.

Query with a Smart SQL SELECT Statement

`buildSmartQuerySpec(smartSql, [pageSize])` executes the query specified by `smartSql`. This function allows greater flexibility than other query factory functions because you provide your own Smart SQL SELECT statement. See [Smart SQL Queries](#).

`pageSize` is optional and defaults to 10.

Sample code, in various development environments, for 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
});
```

iOS native:

```
SFQuerySpec* querySpec = [SFQuerySpec newSmartQuerySpec:@"select count(*) from {employees}"
  withPageSize:1];
NSArray* result = [_store queryWithQuerySpec:querySpec pageIndex:0];
// result should be [[ n ]] if there are n employees
```

Android native:

```
SmartStore store = SalesforceSDKManagerWithSmartStore.getInstance().getSmartStore();
JSONArray result = store.query(QuerySpec.buildSmartQuerySpec("select count(*) from
{employees}", 1), 0);
// result should be [[ n ]] if there are n employees
```

Query by Exact

`buildExactQuerySpec(indexPath, matchKey, [pageSize])` finds entries that exactly match the given `matchKey` for the `indexPath` value. Use this to find child entities of a given ID. For example, you can find Opportunities by Status. However, you can't specify order in the results.

Sample code for retrieving 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]
});
```

Sample code for retrieving children by parent ID:

```
var querySpec = navigator.smartstore.buildExactQuerySpec("parentSfdcId", "some-sfdc-id");
navigator.smartstore.querySoup("Catalogs", querySpec, function(cursor) {});
```

Query by Range

`buildRangeQuerySpec(indexPath, beginKey, endKey, [order, pageSize])` 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.

`order` and `pageSize` are optional, and default to ascending and 10, respectively. You can specify:

- `buildRangeQuerySpec(indexPath, beginKey, endKey)`
- `buildRangeQuerySpec(indexPath, beginKey, endKey, order)`
- `buildRangeQuerySpec(indexPath, beginKey, endKey, order, pageSize)`

However, you can't specify `buildRangeQuerySpec(indexPath, beginKey, endKey, pageSize)`.

By passing null values to `beginKey` and `endKey`, you can perform open-ended searches:

- Passing null to `endKey` finds all records where the field at `indexPath` is \geq `beginKey`.
- Passing null to `beginKey` finds all records where the field at `indexPath` is \leq `endKey`.
- Passing null to both `beginKey` and `endKey` is the same as querying everything.

Query by Like

`buildLikeQuerySpec(indexPath, likeKey, [order, pageSize])` finds entries whose `indexPath` values are like the given `likeKey`. You can use "foo%" to search for terms that begin with your keyword, "%foo" to search for terms that end with your keyword, and "%foo%" to search for your keyword anywhere in the `indexPath` value. Use this function for general searching and partial name matches. `order` and `pageSize` are optional, and default to ascending and 10, respectively.



Note: Query by Like is the slowest of the query methods.

Executing the Query

Queries run asynchronously and return a cursor to your JavaScript callback. Your success callback should be of the form `function(cursor)`. Use the `querySpec` parameter to pass your query specification to the `querySoup` method.

```
navigator.smartstore.querySoup(soupName, querySpec, successCallback, errorCallback);
```

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. Soup entries can be looked up by `_soupEntryId` by using the `retrieveSoupEntries` method. Note that the return order is not guaranteed, and if entries have been deleted they will be missing from the resulting array. This method 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)
```

Smart SQL Queries

Beginning with Salesforce Mobile SDK version 2.0, 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.

Smart SQL Restrictions

Smart SQL supports only SELECT statements and only indexed paths.

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}
```

Aggregate Functions

Smart SQL support 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}
```

The NativeSqlAggregator sample app delivers a fully implemented native implementation of SmartStore, including Smart SQL support for aggregate functions and joins.

Working With Cursors

Queries can potentially have long result sets that are too large to load. Instead, only a small subset of the query results (a single page) is copied from the native realm to the JavaScript realm at any given time. When you perform a query, a cursor object is returned from the native realm that provides a way to page through a list of query results. The JavaScript code can then move forward and back through the pages, causing pages to be copied to the JavaScript realm.



Note: For advanced users: Cursors are not snapshots of data; they are dynamic. If you make changes to the soup and then start paging through the cursor, you will see those changes. 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. Thus, newly created soup entries can be returned (assuming they satisfy the original query).

Use the following cursor functions to navigate the results of a query:

- `navigator.smartstore.moveCursorToPageIndex(cursor, newPageIndex, successCallback, errorCallback)`—Move the cursor to the page index given, where 0 is the first page, and the last page is defined by `totalPages - 1`.

- `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: `successCallback` for those functions should expect one argument (the updated cursor).

Manipulating Data

In order to track soup entries for insert, update, and delete, 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`—The number of milliseconds since 1/1/1970.
 - ◊ To convert 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 inserting or updating soup entries, SmartStore automatically sets these fields. When removing or retrieving specific entries, you can reference them by `_soupEntryId`.

Inserting or Updating Soup Entries

If the provided soup entries already have the `_soupEntryId` slots set, then entries identified by that slot are updated in the soup. If an entry does not have a `_soupEntryId` slot, or the value of the slot doesn't match any existing entry in the soup, then the entry is added (inserted) to the soup, and the `_soupEntryId` slot is overwritten.



Note: You must not manipulate the `_soupEntryId` or `_soupLastModifiedDate` value yourself.

Use the `upsertSoupEntries` method to insert or update entries:

```
navigator.smartstore.upsertSoupEntries(soupName, entries[], successCallback, errorCallback)
```

where `soupName` is the name of the target soup, and `entries` is an array of one or more entries that match the soup's data structure. The `successCallback` and `errorCallback` parameters function much like the ones for `registerSoup`. However, the success callback for `upsertSoupEntries` indicates that either a new record has been inserted, or an existing record has been updated.

Upserting with an External ID

If your soup entries mirror data from an external system, you might need to refer to those entities by their ID (primary key) in the external system. For that purpose, we support upsert with an external ID. When you perform an upsert, you can designate any index field as the external ID field. SmartStore will look 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, a new soup entry will be created.
- If the external ID field is found, it will be updated.
- If more than one field matches the external ID, an error will be returned.

When creating a new entry locally, use a regular upsert. Set the external ID field to a value that you can later query when uploading the new entries to the server.

When updating entries with data coming from the server, use the upsert with external ID. Doing so guarantees that you don't end up with duplicate soup entries for the same remote record.

In the following sample code, we chose the value `new` for the `id` field because the record doesn't yet exist on the server. Once we are online, we 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 the actual ID for the records, we can update their `id` fields locally. If you create products that belong to catalogs that have not yet been created on the server, you will be able to capture the relationship with the catalog through the `parentSoupEntryId` field. Once the catalogs are created on the server, update the local records' `parentExternalId` fields.

The following code contains sample scenarios. First, it calls `upsertSoupEntries` to create a new soup entry. In the success callback, the code retrieves the new record with its newly assigned soup entry ID. It then changes the description and calls `forcetk.mobilesdk` methods to create the new account on the server and then update it. The final call demonstrates the upsert with external ID. To make the code more readable, no error callbacks are specified. Also, because all SmartStore calls are asynchronous, real applications should do each step in the callback of the previous step.

```
// 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 acc has no _soupEntryId field
navigator.smartstore.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
navigator.smartstore.upsertSoupEntries("accounts", [ acc ], function(accounts) {
    acc = accounts[0];
});

// Create account on server (sync client -> server for entities created locally)
forcetkClient.create("account", {"Name": acc["Name"], "Description": acc["Description"]},
    function(result) {
        acc["id"] = result["id"];
        // Update account in SmartStore
        navigator.smartstore.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
forcetkClient.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
forcetkClient.retrieve("account", someSfdcId, "id,Name,Description", function(result) {
    // Create or update account in SmartStore (looking for an account with the same sfdcId)

    navigator.smartstore.upsertSoupEntriesWithExternalId("accounts", [ result ], "id");
});
```

Removing Soup Entries

Entries are removed from the soup asynchronously and your callback is called with success or failure. The `soupEntryIds` is a list of the `_soupEntryId` values from the entries you wish to delete.

```
navigator.smartStore.removeFromSoup(soupName, soupEntryIds, successCallback, errorCallback)
```

Removing a Soup

To remove a soup, call `removeSoup()`. Note that once a user signs out, the soups get deleted automatically.

```
navigator.smartstore.removeSoup(soupName, successCallback, errorCallback);
```

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 the 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 meant only for testing plugins outside the container. Intercepts Cordova plugin calls
- `MockSmartStore.js`—A JavaScript implementation of the SmartStore meant only for development and testing outside the container. Also intercepts SmartStore Cordova plugin 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-x.x.x.js`.
- Include `MockSmartStore.js` after `cordova.force.js`.

To see a MockSmartStore example, check out `external/shared/test/test.html`.

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 Force.com 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 `forcetk.mobilesdk` 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.

NativeSqlAggregator Sample App: Using SmartStore in Native Apps

The NativeSqlAggregator app demonstrates how to use SmartStore in a native app. It also demonstrates the ability to make complex SQL-like queries, including aggregate functions, such as SUM and COUNT, and joins across different soups within SmartStore.

Creating a Soup

The first step to storing a Salesforce object in SmartStore is to create a soup for the object. The function call to register a soup takes two arguments—the name of the soup, and the index specs for the soup. Indexing supports three types of data: string, integer, and floating decimal. The following example illustrates how to initialize a soup for the Account object with indexing on Name, Id, and OwnerId fields.

Android:

```
SalesforceSDKManagerWithSmartStore sdkManager =
SalesforceSDKManagerWithSmartStore.getInstance();

SmartStore smartStore = sdkManager.getSmartStore();

IndexSpec[] ACCOUNTS_INDEX_SPEC = {
    new IndexSpec("Name", Type.string),
    new IndexSpec("Id", Type.string),
    new IndexSpec("OwnerId", Type.string)
};

smartStore.registerSoup("Account", ACCOUNTS_INDEX_SPEC);
```

iOS:

```
SFSmartStore *store = [SFSmartStore sharedStoreWithName:kDefaultSmartStoreName];

NSArray *keys = [NSArray arrayWithObjects:@"path", @"type", nil];
NSArray *nameValues = [NSArray arrayWithObjects:@"Name", kSoupIndexTypeString, nil];
NSDictionary *nameDictionary = [NSDictionary dictionaryWithObjects:nameValues
forKeys:keys];
NSArray *idValues = [NSArray arrayWithObjects:@"Id", kSoupIndexTypeString, nil];
NSDictionary *idDictionary = [NSDictionary dictionaryWithObjects:idValues forKeys:keys];
NSArray *ownerIdValues = [NSArray arrayWithObjects:@"OwnerId", kSoupIndexTypeString,
```



```

nil];
NSDictionary *ownerIdDictionary = [NSDictionary dictionaryWithObjects:ownerIdValues
forKeys:keys];
NSArray *accountIndexSpecs = [[NSArray alloc] initWithObjects:nameDictionary,
idDictionary, ownerIdDictionary, nil];

[store registerSoup:@"Account" withIndexSpecs:accountIndexSpecs];

```

Storing Data in a Soup

Once the soup is created, the next step is to store data in the soup. In the following example, `account` represents a single record of the object `Account`. On Android, `account` is of type `JSONObject`. On iOS, its type is `NSDictionary`.

Android:

```

SmartStore smartStore = sdkManager.getSmartStore();
smartStore.upsert("Account", account);

```

iOS:

```

SFSmartStore *store = [SFSmartStore sharedStoreWithName:kDefaultSmartStoreName];
[store upsertEntries:[NSArray arrayWithObject:account] toSoup:@"Account"];

```

Running Queries Against SmartStore

Beginning with Mobile SDK 2.0, you can run advanced SQL-like queries against SmartStore that span multiple soups. The syntax of a SmartStore query is similar to standard SQL syntax, with a couple of minor variations. A colon (":") serves as the delimiter between a soup name and an index field. A set of curly braces encloses each `<soup-name>:<field-name>` pair. See [Smart SQL Queries](#).

Here's an example of an aggregate query run against SmartStore:

```

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}

```

This query represents an implicit join between two soups, `Account` and `Opportunity`. It returns:

- Name of the Account
- Number of opportunities associated with an Account
- Sum of all the amounts associated with each Opportunity of that Account
- Average amount associated with an Opportunity of that Account
- Grouping by Account name

The code snippet below demonstrates how to run such queries from within a native app. In this example, `smartSql` is the query and `pageSize` is the requested page size. The `pageIndex` argument specifies which page of results you want returned.

Android:

```
QuerySpec querySpec = QuerySpec.buildSmartQuerySpec(smartSql, pageSize);
JSONArray result = smartStore.query(querySpec, pageIndex);
```

iOS:

```
SFSmartStore *store = [SFSmartStore sharedStoreWithName:kDefaultSmartStoreName];
SFQuerySpec *querySpec = [SFQuerySpec newSmartQuerySpec:queryString
withPageSize:pageSize];
NSArray *result = [store queryWithQuerySpec:querySpec pageIndex:pageIndex];
```

Using SmartSync to Access Salesforce Objects

The SmartSync Data Framework is a Mobile SDK library that represents Salesforce objects as JavaScript objects. Using SmartSync in a hybrid app, you can 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.

Underlying the SmartSync technology is the `backbone.js` open-source JavaScript library. `Backbone.js` defines an extensible mechanism for modeling data. To understand the basic technology behind the SmartSync Data Framework, browse the examples and documentation at backbonejs.org.

Three sample hybrid applications demonstrate SmartSync.

- Account Editor (`AccountEditor.html`)
- User Search (`UserSearch.html`)
- User and Group Search (`UserAndGroupSearch.html`)

You can find these sample apps in the `./hybrid/SampleApps/AccountEditor/assets/www` folder.

About Backbone Technology

The SmartSync library, `SmartSync.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 SmartSync Data Framework extends the `Model` and `Collection` core Backbone objects to connect them to the Salesforce REST API. SmartSync 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 SmartSync 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 SmartSync, 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`:

subjectType

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 SmartSync Data Framework 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.subjectType = "account";
acc.fieldlist = ["Id", "Name"];
acc.fetch();
```

```
// As methods on a Force.SObject sub-class
Account = Force.SObject.extend({
  subjectType: "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.subjectType = "account";
acc.fetch({fieldlist:["Id", "Name"]});
```

Model Collections

Model collections in the SmartSync Data Framework 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 SmartSync Data Framework 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 SmartSync Data Framework in JavaScript

To use SmartSync in a hybrid app, include:

- jquery-x.x.x.min.js (use version of file in external/shared/jquery/)
- underscore-x.x.x.min.js (use version of file in external/shared/backbone/)
- backbone-x.x.x.min.js (use version of file in external/shared/backbone/)
- cordova.js
- cordova.force.js
- forcetk.mobilesdk.js
- SmartSync.js

Implementing a Model Object

To begin using SmartSync 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—`objectType` and `fieldlist`—and defines a `cacheMode()` function.

```
app.models.Account = Force.SObject.extend({
  objectType: "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 `SmartSync.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.

SmartSync 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", objectType:"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. SmartSync provides a configurable mechanism that gives you full control over caching operations.

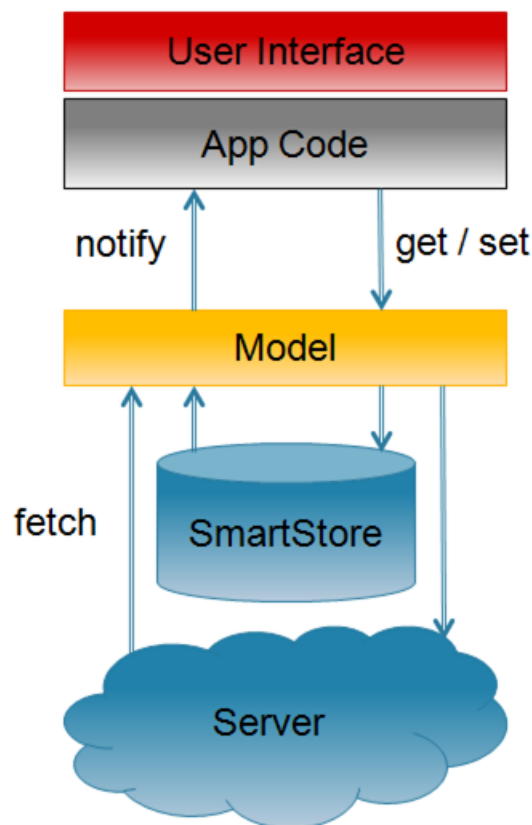
Default Cache and Custom Cache Implementations

For its default cache, the SmartSync library defines `StoreCache`, a cache implementation that uses `SmartStore`. Both `StoreCache` and `SmartStore` are optional components for SmartSync 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. SmartSync requires cache objects to support these operations:

- retrieve
- save
- save all
- remove
- find

SmartSync Caching Workflow

The SmartSync 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.



SmartSync 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"	<code>Force.CACHE_MODE.CACHE_ONLY</code>	Read from, or write to, the cache. Do not perform the operation on the server.
"server-only"	<code>Force.CACHE_MODE.SERVER_ONLY</code>	Read from, or write to, the server. Do not perform the operation on the cache.

Mode	Constant	Description
"cache-first"	<code>Force.CACHE_MODE.CACHE_FIRST</code>	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)	<code>Force.CACHE_MODE.SERVER_FIRST</code>	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, SmartSync 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 `SmartSync.js` library implements a cache named `StoreCache` that stores its data in `SmartStore`. Although `SmartSync` uses `StoreCache` as its default cache, `StoreCache` is a stand-alone component. Even if you don't use `SmartSync`, you can still leverage `StoreCache` for `SmartStore` operations.



Note: Although `StoreCache` is intended for use with `SmartSync`, you can use any cache mechanism with `SmartSync` 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. `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
<code>records</code>	All fetched records
<code>hasMore</code>	Function to check if more records can be retrieved

Field	Description
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. SmartSync 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
overwrite	<code>Force.MERGE_MODE.OVERWRITE</code>	Write "yours" to the server, without comparing to "theirs" or "base". (This is the same as not using conflict detection.)
merge-accept-yours	<code>Force.MERGE_MODE.MERGE_ACCEPT_YOURS</code>	Merge "theirs" and "yours". If the same field is changed both locally and remotely, the local value is kept.
merge-fail-if-conflict	<code>Force.MERGE_MODE.MERGE_FAIL_IF_CONFLICT</code>	Merge "theirs" and "yours". If the same field is changed both locally and remotely, the operation fails.
merge-fail-if-changed	<code>Force.MERGE_MODE.MERGE_FAIL_IF_CHANGED</code>	Merge "theirs" and "yours". If any field is changed remotely, the operation fails.

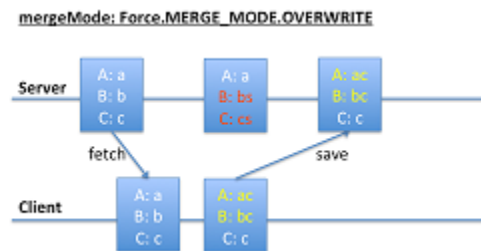
If a save or delete operation fails, you receive a report object with the following fields:

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.

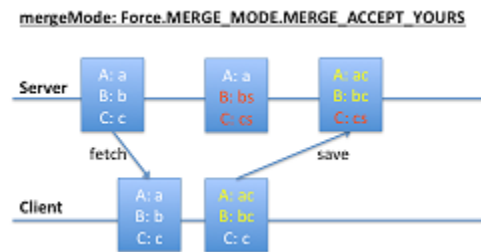
MERGE_MODE.OVERWRITE

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.



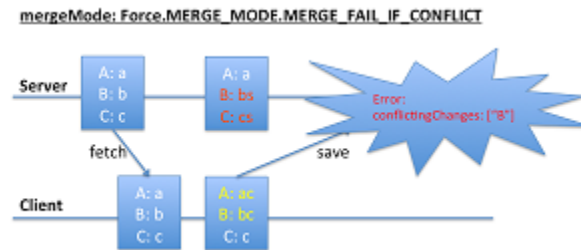
MERGE_ACCEPT_YOURS

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.



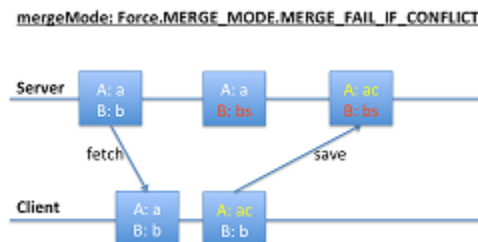
MERGE_FAIL_IF_CONFLICT (Fails)

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({subjectType:"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, SmartSync expands its scope to let you work with any REST API. Previously, you could only perform basic operations on sObjects with the Force.com API. Now you can use SmartSync 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, SmartSync 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 Force.com REST API. If you want to use other server endpoints, 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

The `HybridFileExplorer` sample application is a SmartSync app that shows how to use `Force.RemoteObject`. `HybridFileExplorer` calls the Chatter 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 `forcetk.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.forcetkClient.fileDetails(id, null);
  }
});
```

Force.RemoteObjectCollection Class

To support collections of fetched objects, SmartSync 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 Force.com REST API to run SOQL/SOSL and MRU queries. If you want to use arbitrary server endpoints, 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

The `HybridFileExplorer` sample application also demonstrates how to use `Force.RemoteObjectCollection`. This example calls the Chatter 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.forcetkClient.ownedFilesList("me", 0);
        break;
      case "filesInUsersGroups": fetchPromise =
        Force.forcetkClient.filesInUsersGroups("me", 0);
        break;
      case "filesSharedWithUser": fetchPromise =
        Force.forcetkClient.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
            forcetkClient.queryMore(nextPageUrl)
              .then(function(resp) {
                nextPageUrl = resp.nextPageUrl;
                that.records.pushObjects(resp.files);
              });
        }
      };
    });
}
```

Using Apex REST Resources

Force.ApexRestObject

Example

216

```

        return new Map<String, String>{
            '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 SmartSync, 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:"<valid id>"})
acc2.fetch();

```

You can delete a "simple account".

```

acc.destroy();

```



Note: In SmartSync 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

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 SmartSync, you do the following to fetch a list of "simple account" records.

```
var SimpleAccountCollection =
    Force.ApexRestObjectCollection.extend(
        {model: SimpleAccount,
        config:{
            apexRestPath:"/simpleAccounts",
            params:{namePattern:"My%"}
        }
    });
var accs = new SimpleAccountCollection();
accs.fetch();
```



Note: In SmartSync calls such as `fetch()`, you typically pass an options parameter that defines success and error callback functions. For example:

```
acc.fetch({success:function(){alert("fetched " +
    accs.models.length + " simple accounts");}});
```

Using External Objects (Beta)



Note: Any unreleased services or features referenced in this document, press releases, or public statements are not currently available and may not be delivered on time or at all. Customers who purchase our services should make their purchase decisions based upon features that are currently available.

About External Objects (Beta)

If you have data stored outside of Salesforce, you might need to use it with data inside your organization. For example, you might need to access inventory information that resides in an external database to more easily reconcile your stock. Salesforce lets you connect to an external data source from within your organization, access the data you want, create an external object for the data, and make it accessible to specific users from a tab. You can learn more about defining external objects at http://help.salesforce.com/apex/HTViewHelpDoc?id=external_object_manage.htm.



Note: The Salesforce API only lets you read external objects. CRUD operations are not supported.

Accessing External Objects (Beta) in Mobile SDK Apps

To access an external object with Mobile SDK, use `SmartSync.js`. Create an instance of either `Force.SObject` itself or a subclass of `Force.SObject`. Configure this instance as follows:

- Set `idAttribute` to "ExternalId".
- If you use a cache, set `idField` to "ExternalId".

Example: Fetch an External Object Using SmartSync

The following JavaScript example accesses an external object named `Categories__x` with a field named `CategoryName__c` by extending `Force.SObject`.

1. Set up a cache using `StoreCache`.

```
var cache = new Force.StoreCache("categories",
    ["CategoryName__c"], "ExternalId");
cache.init();
```

2. Create a `Force.SObject` subclass to represent `Categories__x` objects on the client. Set `idAttribute` to "ExternalId", and pass in `cache` to enable caching.

```
var Category = Force.SObject.extend({
    objectType: "Categories__x",
    idAttribute: "ExternalId",
    fieldlist: ["CategoryName__c"],
    cache: cache});
```

3. Create an instance of `Category` that can fetch an external object whose external ID is "1", then fetch that object.

```
var cat = new Category({"ExternalId": "1"});
cat.fetch();
```

4. Retrieve the fetched object from the cache:

```
var cat1 = cache.retrieve("1");
```

Example: Fetch a Collection of External Objects Using SmartSync

You can also use a `Force.SObjectCollections` object to represent a collection of `Category` objects on the client.

1. Create a subclass of `Force.SObjectCollection`.

```
var Categories = Force.SObjectCollection.extend({model:Category, cache:cache});
```

2. Fetch `Categories__x` objects by running a SOQL query.

```
var categories = new Categories();
categories.fetch({config:{
  type:"soql",
  query:"SELECT ExternalId, CategoryName__c
        FROM Categories__x"}
});
```

3. Retrieve the fetched object whose external ID is “2” from the collection within the cache.

```
var cat2 = cache.retrieve("2");
```

Tutorial: Creating a SmartSync Application

This tutorial demonstrates how to create a local hybrid app that uses the SmartSync Data Framework. It recreates the User Search sample application that ships with Mobile SDK 2.0. User Search 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

Set Up Your Project

First, make sure you’ve installed Salesforce Mobile SDK using the NPM installer. For iOS instructions, see [iOS Installation](#). For Android instructions, see [Android Installation](#).

Also, download the `ratchet.css` file from <http://maker.github.io/ratchet/>.

1. Once you’ve installed Mobile SDK, create a local hybrid project for your platform.
 - a. For **iOS**: At the command terminal, enter the following command:

```
forceios create --apptype=hybrid_local
--appname=UserSearch --companyid=com.acme.UserSearch
--organization=Acme --outputdir=.
```

The `forceios` script creates your project at `./UserSearch/UserSearch.xcode.proj`.

- b. For **Android**: At the command terminal or the Windows command prompt, enter the following command:

```
forcedroid create --apptype="hybrid_local"
--appname="UserSearch" --targetdir=.
--packagename="com.acme.usersearch"
```

The forcedroid script creates the project at `./UserSearch`.

2. Follow the onscreen instructions to open the new project in Eclipse (for Android) or Xcode (for iOS).
3. Open the `www` folder.
4. Remove the `inline.js` file from the project.
5. Create a new folder. Name it `css`.
6. Copy the `ratchet.css` file into your new `css` folder.
7. In the `www` folder, open `index.html` in your code editor and delete all of 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. In Xcode, edit `index.html` and add the following basic structure:

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
  </body>
</html>
```

2. In the `<head>` element:

- a. 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;" />
```

- b. Set the content type.

```
<meta http-equiv="Content-type" content="text/html; charset=utf-8">
```

- c. Add a link to the `ratchet.css` file to provide the mobile look:

```
<link rel="stylesheet" href="css/ratchet.css"/>
```

- d. Include the necessary JavaScript files.

```
<script src="jquery/jquery-2.0.0.min.js"></script>
<script src="backbone/underscore-1.4.4.min.js"></script>
<script src="backbone/backbone-1.0.0.min.js"></script>
<script src="cordova-2.3.0.js"></script>
<script src="forcetk.mobilesdk.js"></script>
<script src="cordova.force.js"></script>
<script src="SmartSync.js"></script>
```

- Now let's start adding content to the body. In the `<body>` block, add a `div` tag to contain the app UI.

```
<body>
<div id="content"></div>
```

It's good practice to keep your objects and classes in a namespace. In this sample, we use the `app` namespace to contain our models and views.

- In a `<script>` tag, create an application namespace. Let's call it `app`.

```
<script>
var app = {
  models: {},
  views: {}
}
```

For the remainder of this procedure, continue adding your code in the `<script>` block.

- Add an event listener and handler to wait for jQuery, and then call Cordova to start the authentication flow. Also, specify a callback function, `appStart`, to handle the user's credentials.

```
jQuery(document).ready(function() {
  document.addEventListener("deviceready", onDeviceReady, false);
});

function onDeviceReady() {
  cordova.require("salesforce/plugin/oauth").
    getAuthCredentials(appStart);
}
```

Once the application has initialized and authentication is complete, the Salesforce OAuth plugin calls `appStart()` and passes it the user's credentials. The `appStart()` function passes the credentials to SmartSync by calling `Force.init()`, which initializes SmartSync. The `appStart()` function also creates a Backbone Router object for the application.

- Add the `appStart()` function definition at the end of the `<script>` block.

```
function appStart(creds) {
  Force.init(creds, null, null,
    cordova.require("salesforce/plugin/oauth").forcetkRefresh);
  app.router = new app.Router();
  Backbone.history.start();
}
```

Here's the complete application to this point.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0,
      maximum-scale=1.0; user-scalable=no" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <link rel="stylesheet" href="css/ratchet.css"/>
    <script src="jquery/jquery-2.0.0.min.js"></script>
    <script src="backbone/underscore-1.4.4.min.js"></script>
    <script src="backbone/backbone-1.0.0.min.js"></script>
    <script src="cordova-2.3.0.js"></script>
    <script src="forcetk.mobilesdk.js"></script>
```



```

    <script src="cordova.force.js"></script>
    <script src="SmartSync.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <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>

    <script id="user-list-item" type="text/template">
      
      <div class="details-short">
        <b><%= FirstName %> <%= LastName %></b><br/>
        Title<%= Title %>
      </div>
    </script>

    <script>
    var app = {
      models: {},
      views: {}
    };

    jQuery(document).ready(function() {
      document.addEventListener("deviceready", onDeviceReady, false);
    });

    function onDeviceReady() {
      cordova.require("salesforce/plugin/oauth").
        getAuthCredentials(appStart);
    }

    function appStart(creds) {
      console.log(JSON.stringify(creds));
      Force.init(creds, null, null,
        cordova.require("salesforce/plugin/oauth").forcetkRefresh);
      app.router = new app.Router();
      Backbone.history.start();
    }
  </script>
</body>
</html>

```

Create a SmartSync Model and a Collection

Now that we've configured the HTML infrastructure, let's get started using SmartSync 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 model object for the Salesforce User sObject. Extend `Force.SObject` to specify the sObject type and the fields we are targeting.

```
app.models.User = Force.SObject.extend({
  objectType: "User",
  fieldlist: ["Id", "FirstName", "LastName", "SmallPhotoUrl", "Title", "Email",
    "MobilePhone", "City"]
});
```

2. Immediately after setting the User object, create a collection to hold user search results. Extend `Force.SObjectCollection` to indicate your new model (`app.models.User`) as the model for items in the collection.

```
app.models.UserCollection = Force.SObjectCollection.extend({
  model: app.models.User
});
```

Here's the complete model code.

```
// Models
app.models.User = Force.SObject.extend({
  objectType: "User",
  fieldlist: ["Id", "FirstName", "LastName", "SmallPhotoUrl", "Title", "Email",
    "MobilePhone", "City"]
});

app.models.UserCollection = Force.SObjectCollection.extend({
  model: app.models.User
});
```

Create a Template

Templates let you describe an HTML layout within another HTML page. You can define an inline template in your HTML page by using a `<script>` tag of type `"text/template"`. Your JavaScript code can use the template as 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.

1. Add a new script block. Place the block within the `<body>` block just 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>
```

Add the Search View

To create the view for a screen, you extend `Backbone.View`. In the search view extension, you load the template, define sub-views and event handlers, and implement the functionality for rendering the views and performing a SOQL search query.

1. In the `<body>` block, 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.

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. Instantiate a sub-view named `UserListView` to contain the list of search results. (You'll define the `app.views.UserListView` view later.)

```
initialize: function() {
  this.listView = new app.views.UserListView({model: this.model});
},
```

4. Create a `render()` function for the search page view. Rendering the view consists simply of loading the template as the app's HTML content. Restore any criteria previously typed in the search field and render the sub-view inside the `` element.

```
render: function(eventName) {
  $(this.el).html(this.template());
  $(".search-key", this.el).val(this.model.criteria);
  this.listView.setElement($(".ul", this.el)).render();
  return this;
},
```

5. Add a `keyup` event handler that performs a search when the user types a character in the search field.

```
events: {
  "keyup .search-key": "search"
},

search: function(event) {
  this.model.criteria = $(".search-key", this.el).val();
  var soql = "SELECT Id, FirstName, LastName, SmallPhotoUrl, Title FROM User WHERE Name like '" + this.model.criteria + "%' ORDER BY Name LIMIT 25 ";
  this.model.fetch({config: {type:"soql", query:soql}});
}
```

This function defines a SOQL query. It then uses the backing model to send that query to the server and fetch the results.

Here's the complete extension.

```
app.views.SearchPage = Backbone.View.extend({
  template: _.template($("#search-page").html()),

  initialize: function() {
    this.listView = new app.views.UserListView({model: this.model});
  },

  render: function(eventName) {
    $(this.el).html(this.template());
    $(".search-key", this.el).val(this.model.criteria);
    this.listView.setElement($("#ul", this.el)).render();
    return this;
  },

  events: {
    "keyup .search-key": "search"
  },

  search: function(event) {
    this.model.criteria = $(".search-key", this.el).val();
    var soql = "SELECT Id, FirstName, LastName, SmallPhotoUrl, Title FROM User WHERE
Name like '" + this.model.criteria + "%' ORDER BY Name LIMIT 25 ";
    this.model.fetch({config: {type:"soql", query:soql}});
  }
});
```

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 keeps track of these views in the `listItemViews` member. If the underlying collection changes, it renders itself again.

1. In the `<body>` block, create the view for the search result list by extending `Backbone.View`. Let's add an array for list item views as well as 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 to clean up any existing list item views by calling `close()` on each one.

```
render: function(eventName) {
  _.each(this.listItemViews, function(itemView) { itemView.close(); });
```

3. In the `render()` function, create a new set of list item views for the records in the underlying collection. Each of these views is just an entry in the list. You'll define the `app.views.UserListItemView` later.

```
this.listItemViews = _.map(this.model.models, function(model) { return new
  app.views.UserListItemView({model: model}); });
```

4. Append the list item views to the root DOM element.

```
$(this.el).append(_.map(this.listItemViews, function(itemView) {
  return itemView.render().el; } ));
return this;
}
```

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. In the `<body>` block, create a template for a search result list item.

```
<script id="user-list-item" type="text/template">
  
  <div class="details-short">
    <b><%= FirstName %> <%= LastName %></b><br/>
    Title<%= Title %>
  </div>
</script>
```

2. Immediately after the template, create the view for the search result list item. Once again, subclass `Backbone.View` and indicate that the whole view should be rendered as a list by defining the `tagName` member. For the remainder of this procedure, add all code in the `extend({})` block.

```
app.views.UserListItemView = Backbone.View.extend({
  tagName: "li",
});
```

3. Load template by calling `_.template()` with the raw content of the `user-list-item` script.

```
template: _.template($("#user-list-item").html()),
```

4. In the `render()` function, simply render the template using data from the model.

```
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 to do necessary cleanup and avoid memory leaks.

```
close: function() {
  this.remove();
  this.off();
}
```

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();
  }
});
```

Router

A Backbone router defines navigation paths among views. To learn more about routers, see [What is a router?](#)

1. Just before the closing tag of the `<body>` block, define the application router by extending `Backbone.Router`.

```
app.Router = Backbone.Router.extend({
});
```

For the remainder of this procedure, add all code in the `extend({})` block.

2. Because the app supports only one screen, you need only one “route”. Add a `routes` object.

```
routes: {
  "": "list"
},
```

3. Define an `initialize()` function that creates the search result collections and search page view.

```
initialize: function() {
  Backbone.Router.prototype.initialize.call(this);

  // Collection behind search screen
  app.searchResults = new app.models.UserCollection();
  app.searchView = new app.views.SearchPage({model: app.searchResults});
},
```

4. Define the `list()` function to handle the only item in this route. When the list screen displays, fetch the search results and render the search view.

```
list: function() {
  app.searchResults.fetch();
  $('#content').html(app.searchView.render().el);
}
```

5. Run the application by double-clicking `index.html` to open it in a browser.

You’ve finished! Here’s the entire application:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,
      initial-scale=1.0, maximum-scale=1.0; user-scalable=no" />
    <meta http-equiv="Content-type" content="text/html;
      charset=utf-8">
    <link rel="stylesheet" href="css/ratchet.css"/>
    <script src="jquery/jquery-2.0.0.min.js"></script>
    <script src="backbone/underscore-1.4.4.min.js"></script>
    <script src="backbone/backbone-1.0.0.min.js"></script>
    <script src="cordova-2.3.0.js"></script>
    <script src="forcetk.mobilesdk.js"></script>
    <script src="cordova.force.js"></script>
    <script src="SmartSync.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <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>
  </body>
</html>
```

```

    </div>
</script>

<script id="user-list-item" type="text/template">
  
  <div class="details-short">
    <b><%= FirstName %> <%= LastName %></b><br/>
    Title<%= Title %>
  </div>
</script>

<script>
var app = {
  models: {},
  views: {}
};

jQuery(document).ready(function() {
  document.addEventListener("deviceready", onDeviceReady, false);
});

function onDeviceReady() {
  cordova.require("salesforce/plugin/oauth").
    getAuthCredentials(appStart);
}

function appStart(creds) {
  console.log(JSON.stringify(creds));
  Force.init(creds, null, null,
    cordova.require("salesforce/plugin/oauth").forcetkRefresh);
  app.router = new app.Router();
  Backbone.history.start();
}

// Models
app.models.User = Force.SObject.extend({
  objectType: "User",
  fieldlist: ["Id", "FirstName", "LastName",
    "SmallPhotoUrl", "Title", "Email", "MobilePhone",
    "City"]
});

app.models.UserCollection = Force.SObjectCollection.extend({
  model: app.models.User
});

// Views
app.views.SearchPage = Backbone.View.extend({
  template: _.template($("#search-page").html()),

  initialize: function() {
    this.listView =
      new app.views.UserListView({model: this.model});
  },

  render: function(eventName) {
    $(this.el).html(this.template());
    $(".search-key", this.el).val(this.model.criteria);
    this.listView.setElement($("#ul", this.el)).render();
    return this;
  },

  events: {
    "keyup .search-key": "search"
  },

```



```

        search: function(event) {
            this.model.criteria = $(".search-key", this.el).val();
            var soql = "SELECT Id, FirstName, LastName,
                SmallPhotoUrl, Title
                FROM User WHERE Name like '" + this.model.criteria + "%'
                ORDER BY Name LIMIT 25 ";
            this.model.fetch({config: {type:"soql", query:soql}});
        }
    });

    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();
        }

    });

    // Router
    app.Router = Backbone.Router.extend({
        routes: {
            "": "list"
        },

        initialize: function() {
            Backbone.Router.prototype.initialize.call(this);

            // Collection behind search screen
            app.searchResults = new app.models.UserCollection();
            app.searchView =
                new app.views.SearchPage({model: app.searchResults});
            console.log("here");
        },

        list: function() {
            app.searchResults.fetch();
            $('#content').html(app.searchView.render().el);
        }

    });

</script>
</body>
</html>

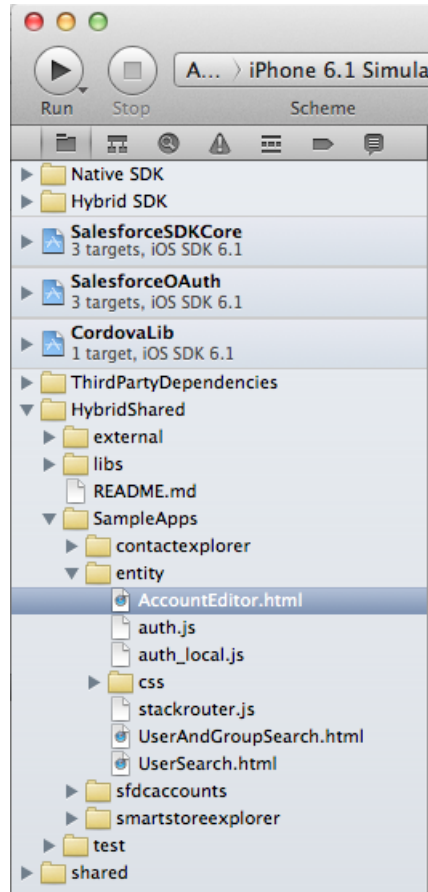
```

SmartSync Sample Apps

Salesforce Mobile SDK provides sample apps that demonstrate how to use SmartSync 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, open `HybridShared/sampleApps/smartsync/AccountEditor.html`.



Running the Samples in Android

In Android, you can run the sample from the command prompt. In your Salesforce Mobile SDK for Android installation directory, change to the `hybrid/SampleApps/AccountEditor` directory and run:

```
ant debug
ant install
```



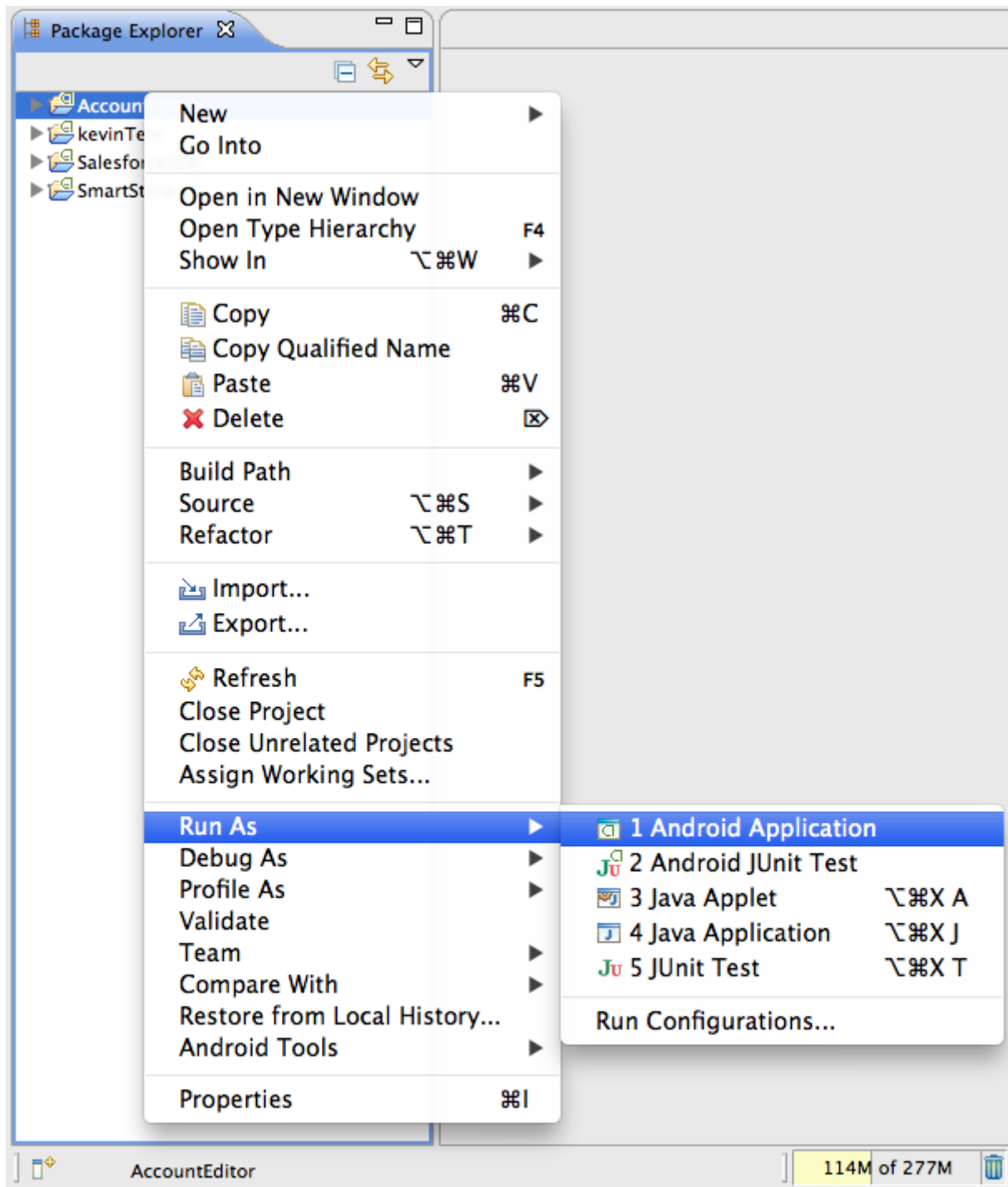
Note: If you get any errors saying that the `local.properties` file does not exist, run the following command from the directory shown in the error message:

```
%ANDROID_SDK%/tools/android update project -p .
```

To run the sample in Eclipse, import the following projects into your workspace:

- forcedroid/native/SalesforceSDK
- forcedroid/hybrid/SmartStore
- forcedroid/hybrid/SampleApps/AccountEditor

After Eclipse finishes building, Control-click or right-click **AccountEditor** in the Package Explorer, then click **Run As > Android application**.



User and Group Search Sample

User and group search is the simplest SmartSync sample app. Its single screen lets you search users and collaboration groups and display matching records in a list.

To run the sample, edit `external/shared/sampleApps/smartsync/bootconfig.json`. Change `startPage` to `UserAndGroupSearch.html`:

```
{
  "remoteAccessConsumerKey":
    "3MVG9Iu66FKeHhINKB1l7xt7kR8czFcCTUhgoA80l2LtflEYHOU4SqQRSEitYFDUpqRWcoQ2.
    dBv_a1Dyu5xa",
  "oauthRedirectURI": "testsfdc:///mobilesdk/detect/oauth/done",
  "oauthScopes": ["api", "web"],
  "isLocal": true,
  "startPage": "UserAndGroupSearch.html",
  "errorPage": "error.html",
  "shouldAuthenticate": true,
  "attemptOfflineLoad": true
  "remoteAccessConsumerKey":
    "3MVG9Iu66FKeHhI2LtflEYHOU4SqQRSEitYFDUpqRWcoQ2.
    dBv_a1Dyu5xa",
  "oauthRedirectURI": "testsfdc:///mobilesdk/detect/oauth/done",
  "oauthScopes": ["api", "web"],
  "isLocal": true,
  "startPage": "UserAndGroupSearch.html",
  "errorPage": "error.html",
  "shouldAuthenticate": true,
  "attemptOfflineLoad": true
  "remoteAccessConsumerKey":
    "3MVG9Iu66FKeHhINKB1l7xt7kR8czFcCTUhgoA80l2LtflEYHOU4SqQRSEitYFDUpqRWcoQ2.
    dBv_a1Dyu5xa",
  "oauthRedirectURI": "testsfdc:///mobilesdk/detect/oauth/done",
  "oauthScopes": ["api", "web"],
  "isLocal": true,
  "startPage": "UserAndGroupSearch.html",
  "errorPage": "error.html",
  "shouldAuthenticate": true,
  "attemptOfflineLoad": true
}
```

To run the app from Xcode in iOS, click **Run** to launch the AccountEditor project. After you've logged in, type at least two characters in the search box to see matching results.

Looking Under the Hood

Open `UserAndGroupSearch.html` in your favorite 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 SmartSync 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 SmartSync Data Framework. See <http://backbonejs.org/>.
- cordova-2.3.0.js—Required for all hybrid application used the SalesforceMobileSDK.
- 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—template for the entire search page
- user-list-item—template for user list items
- group-list-item—template for collaboration group list items

Models

This application defines a `SearchCollection` model.

`SearchCollection` subclasses the `Force.SObjectCollection` class, which in turn subclasses the `Collection` class from the Backbone library. Its only method configures the SOSL query used by the `fetch()` method to populate the collection.

```
app.models.SearchCollection = Force.SObjectCollection.extend({
  setCriteria: function(key) {
    this.config = {type:"sosl", query:"FIND {" + key + "} IN ALL FIELDS RETURNING "
      + "CollaborationGroup (Id, Name, SmallPhotoUrl, MemberCount), "
      + "User (Id, FirstName, LastName, SmallPhotoUrl, Title ORDER BY Name) "
      + "LIMIT 25"
    };
  }
});
```

Views

User and Group Search defines three views:

SearchPage

The search page expects a `SearchCollection` as its model. It watches the search input field for changes and updates the model accordingly.

```
events: {
  "keyup .search-key": "search"
},

search: function(event) {
  var key = $(".search-key", this.el).val();
  if (key.length >= 2) {
    this.model.setCriteria(key);
    this.model.fetch();
  }
}
```

ListView

The list portion of the search screen. `ListView` also expects a `Collection` as its model and creates `ListItemView` objects for each record in the `Collection`.

ListItemView

Shows details of a single list item, choosing the User or Group template based on the data.

Router

The router does very little because this application defines only one screen.

User Search Sample

User Search is a more elaborate sample than User and Group search. Instead of a single screen, it defines two screens. If your search returns a list of matches, User Search lets you tap on each of them to see a basic detail screen. Because it defines more than one screen, this sample also demonstrates the use of a router.

To run the sample, edit `external/shared/sampleApps/smartsync/bootconfig.json`. Change `startPage` to `UserSearch.html`:

```
{
  "remoteAccessConsumerKey": "3MVG9Iu66FKeHhI2Ltf1eYHOU4SqQRSEitYFDUpqRWcoQ2.dBv_a1Dyu5xa",
  "oauthRedirectURI": "testsfdc:///mobilesdk/detect/oauth/done",
  "oauthScopes": ["api", "web"],
  "isLocal": true,
  "startPage": "UserSearch.html",
  "errorPage": "error.html",
  "shouldAuthenticate": true,
  "attemptOfflineLoad": true
}
```

In Xcode or Eclipse, launch the AccountEditor. Log in if prompted to do so. Unlike the User and Group Search example, you need to type only a single character in the search box to begin seeing search results. That's because this application uses SOQL, rather than SOSL, to query the server.

When you tap an entry in the search results list, you see a basic detail screen.

Looking Under the Hood

Open the `UserSearch.html` file in your favorite 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 SmartSync 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 SmartSync Data Framework. See <http://backbonejs.org/>.
- `cordova-2.3.0.js`—Required for all hybrid application used the SalesforceMobileSDK.
- `forcetk.mobilesdk.js`—Force.com JavaScript library for making Rest API calls. Required by SmartSync.

- `cordova.force.js`—As of Mobile SDK 2.0, this file combines all Force.com Cordova plugins. Replaces the `SFHybridApp.js`, `SalesforceOAuthPlugin.js`, and `SmartStorePlugin.js` files.
- `SmartSync.js`—The Mobile SDK SmartSync Data Framework.
- `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`—template for the whole search page
- `user-list-item`—template for user list items
- `user-page`—template for user detail page

Models

This application defines two models: `UserCollection` and `User`.

`UserCollection` subclasses the `Force.SObjectCollection` class, which in turn subclasses the `Collection` class from the Backbone library. Its only method configures the SOQL query used by the `fetch()` method to populate the collection.

```
app.models.UserCollection = Force.SObjectCollection.extend({
  model: app.models.User,
  fieldlist: ["Id", "FirstName", "LastName", "SmallPhotoUrl", "Title"],

  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 "
    };
  }
});
```

`User` subclasses SmartSync's `Force.SObject` class. The `User` model defines:

- An `objectType` field to indicate which type of sObject it represents (`User`, in this case).
- A `fieldlist` field that contains the list of fields to be fetched from the server

Here's the code:

```
app.models.User = Force.SObject.extend({
  objectType: "User",
  fieldlist: ["Id", "FirstName", "LastName", "SmallPhotoUrl", "Title", "Email",
    "MobilePhone", "City"]
});
```

Views

This sample defines four views:

SearchPage

View for the entire search page. It expects a `UserCollection` as its model. It watches the search input field for changes 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();
}
```

UserListView

View for the list portion of the search screen. It also expects a `UserCollection` as its model and creates `UserListItemView` objects for each user in the `UserCollection` object.

UserListItemView

View for a single list item.

UserPage

View for displaying user details.

Router

The router class handles navigation between the app's two screens. This class uses a `routes` field to map those view to router class method.

```
routes: {
  "": "list",
  "list": "list",
  "users/:id": "viewUser"
},
```

The list page calls `fetch()` to fill the search result collections, then 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 user detail page calls `fetch()` to fill the user model, then brings the user detail page into view.

```
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);
    }
  });
}
```

Account Editor Sample

Account Editor is the most complex SmartSync-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/smartsync/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 SmartSync 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 SmartSync Data Framework. See <http://backbonejs.org/>.
- cordova-2.3.0.js—Required for hybrid applications using the Salesforce Mobile SDK.
- forcetk.mobilesdk.js—Force.com JavaScript library for making REST API calls. Required by SmartSync.
- cordova.force.js—As of Mobile SDK 2.0, this file combines all Force.com Cordova plugins. Replaces the SFHybridApp.js, SalesforceOAuthPlugin.js, and SmartStorePlugin.js files.
- SmartSync.js—The Mobile SDK SmartSync Data Framework.
- 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 SmartSync'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 SmartSync'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 `subjectType` 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({
  subjectType: "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();
}
```

AccountListView

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);
}
```

Chapter 8

Push Notifications and Mobile SDK

In this chapter ...

- [About Push Notifications](#)
- [Using Push Notifications in Android](#)
- [Using Push Notifications in iOS](#)

Push notifications from Salesforce help your mobile users stay on top of important developments in their organizations. The Salesforce Mobile Push Notification Service, which becomes generally available in Summer '14, lets you configure and test mobile push notifications before you implement any code. To receive mobile notifications in a production environment, your Mobile SDK app implements the mobile OS provider's registration protocol and then handles the incoming notifications. Mobile SDK minimizes your coding effort by implementing most of the registration tasks internally.

About Push Notifications

With the Salesforce Mobile Push Notification Service, you can develop and test push notifications in native mobile apps. Salesforce Mobile SDK for native iOS and Android apps 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.

Push notification setup 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
- Implementing Apex triggers

OR

Calling the push notification resource of the Chatter REST API

- Modifying code in your Mobile SDK app
- Registering the mobile device at runtime

You're responsible for Apple or Google service configuration, connected app configuration, Apex or Chatter REST API coding, and minor changes to your Mobile SDK app. Salesforce Mobile SDK handles runtime registration transparently.

For a full description of how to set up mobile push notifications for your organization, see the [Salesforce Mobile Push Notifications Implementation Guide](https://help.salesforce.com/help/doc/en/salesforce_mobile_push_notifications_implementation_guide.pdf). After Summer '14 becomes generally available, you can find this document at https://help.salesforce.com/help/doc/en/salesforce_mobile_push_notifications_implementation_guide.pdf.

Using Push Notifications in Android

Salesforce sends push notifications to Android apps through the Google Cloud Messaging for Android (GCM) framework. See <http://developer.android.com/google/gcm/index.html> for an overview of this framework.

When developing an Android app that supports push notifications, remember these key points:

- You must be a member of the Android Developer Program.
- You can test GCM push services only on an Android device with either the Android Market app or Google Play Services installed. Push notifications don't work on an Android emulator.

To begin, create a Google API project for your app. Your project must have the GCM for Android feature enabled. See <http://developer.android.com/google/gcm/gs.html> 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, you'll need to add the GCM 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.

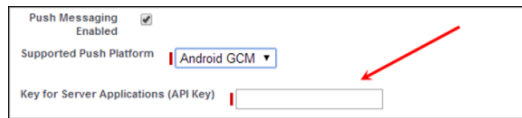
Configure a Connected App For GCM (Android)

To configure your Salesforce connected app to support push notifications:

1. In your Salesforce organization, go to **Setup** > **Create** > **Apps**.
2. In Connected Apps, click **Edit** next to an existing connected app, or **New** to create a new connected app.

If you're creating a new connected app, see [Create a Connected App](#).

3. Under Mobile App Settings, select **Push Messaging Enabled**.
4. For Supported Push Platform, select **Android GCM**.
5. For Key for Server Applications (API Key), enter the key you obtained during the developer registration with Google.



6. Click **Save**.



Note: After saving a new connected app, you'll get a consumer key. Mobile apps use this key as their connection token.

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">35123627573</string>
```

- In `assets/www/bootconfig.json` (for hybrid apps):

```
"androidPushNotificationClientId": "35123627573"
```

This value represents 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);
}

```

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)

To configure your Salesforce connected app to support push notifications with Apple Push Notification Services (APNS):

1. In your Salesforce org, go to **Setup > Create > Apps**.
2. In Connected Apps, either click **Edit** next to an existing connected app, or **New** to create a new connected app. If you're creating a new connected app, see [Create a Connected App](#).
3. Under Mobile App Settings, select **Push Messaging Enabled**.
4. For Supported Push Platform, select **Apple**.

The page expands to show additional settings.

5. Select the Apple Environment that corresponds to your APNS certificate.
6. Add your .p12 file and its password under **Mobile App Settings > Certificate** and **Mobile App Settings > Certificate Password**.



Note: You obtain the values for Apple Environment, Certificate, and Certificate Password when you configure your app with APNS.

7. Click **Save**.

Code Modifications (iOS)

Salesforce Mobile SDK for iOS provides the `SFPushNotificationManager` class to handle push registration. To use it, import `<SalesforceSDKCore/SFPushNotificationManager>`. The `SFPushNotificationManager` class is available as a runtime singleton:

```
[SFPushNotificationManager sharedInstance]
```

This class implements four registration methods:

```
- (void)registerForRemoteNotifications;
- (void)didRegisterForRemoteNotificationsWithDeviceToken:(NSData*) deviceTokenData;
- (BOOL)registerForSalesforceNotifications; // for internal use
- (BOOL)unregisterSalesforceNotifications; // for internal use
```

Mobile SDK calls `registerForSalesforceNotifications` after login and `unregisterSalesforceNotifications` at logout. You call the other two methods from your `AppDelegate` class.

SFPushNotificationManager Example

To configure your AppDelegate class to support push notifications:

1. Register with Apple for push notifications by calling `registerForRemoteNotifications`. Place the call in the `application:didFinishLaunchingWithOptions:` method.

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window =
        [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
    [self initializeAppViewState];

    //
    // Register with APNS for push notifications. Note that,
    // to receive push notifications from Salesforce,
    // you also need to register for Salesforce notifications in the
    // application:didRegisterForRemoteNotificationsWithDeviceToken:
    // method (as demonstrated below.)
    //
    [[SFPushNotificationManager sharedInstance] registerForRemoteNotifications];

    [[SFAuthenticationManager sharedInstance]
        loginWithCompletion:self.initialLoginSuccessBlock
                        failure:self.initialLoginFailureBlock];

    return YES;
}
```

If registration succeeds, Apple passes a device token to the

`application:didRegisterForRemoteNotificationsWithDeviceToken:` method of your AppDelegate class.

2. Forward the device token from Apple to SFPushNotificationManager by calling `didRegisterForRemoteNotificationsWithDeviceToken` on the SFPushNotificationManager shared instance.

```
- (void)application:(UIApplication*)application
    didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)deviceToken
{
    //
    // Register your device token with the push notification manager
    //
    [[SFPushNotificationManager sharedInstance]
        didRegisterForRemoteNotificationsWithDeviceToken:deviceToken];
}
```

3. Register to receive Salesforce notifications through the connected app by calling `registerForSalesforceNotifications`. Make this call only if the access token for the current session is valid.

```
- (void)application:(UIApplication*)application
    didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)deviceToken
{
    //
    // Register your device token with the push notification manager
    //
    [[SFPushNotificationManager sharedInstance]
        didRegisterForRemoteNotificationsWithDeviceToken:deviceToken];
}
```

```
if ([SFAccountManager sharedInstance].credentials.accessToken != nil) {  
    [[SFPushNotificationManager sharedInstance]  
        registerForSalesforceNotifications];  
}}
```

4. Add the following method to log an error if registration with Apple fails.

```
- (void)application:(UIApplication*)application  
    didFailToRegisterForRemoteNotificationsWithError:(NSError*)error  
{  
    NSLog(@"Failed to get token, error: %@", error);  
}
```


Chapter 9

Using Communities With Mobile SDK Apps

In this chapter ...

- [Communities and Mobile SDK Apps](#)
- [Set Up an API-Enabled Profile](#)
- [Set Up a Permission Set](#)
- [Grant API Access to Users](#)
- [Configure the Login Endpoint](#)
- [Branding Your Community](#)
- [Customizing Communities Login](#)
- [Using External Authentication With Communities](#)
- [Example: Configure a Community For Mobile SDK App Access](#)
- [Example: Configure a Community For Facebook Authentication](#)

Salesforce Communities is a social aggregation feature that supersedes the Portal feature of earlier releases. Communities can include up to five million users, with logical zones for sharing knowledge with Ideas, Answers, and Chatter Answers. With proper configuration, your community users can use their community login credentials to access your Mobile SDK app. Communities also leverage Site.com to enable you to brand your community site and login screen.

Communities and Mobile SDK Apps

To enable community 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 community URL.

With Communities, members that you designate can use your Mobile SDK app to access Salesforce. You define your own community login endpoint, and the Communities feature builds a branded community login page according to your specifications. It also lets you choose authentication providers and SAML identity providers from a list of popular choices.

Community membership is determined by profiles and permission sets. To enable community members to use your Mobile SDK app, configure the following:

- Make sure that each community member has the API Enabled permission. You can set this permission through profiles or permission sets.
- Configure your community to include your API-enabled profiles and permission sets.
- Configure your Mobile SDK app to use your community's login endpoint.

In addition to these high-level steps, you must take the necessary steps to configure your users properly. [Example: Configure a Community For Mobile SDK App Access](#) walks you through the community configuration process for Mobile SDK apps. For the full documentation of the Communities feature, see [Getting Started With Communities](#).



Note: Community login is supported for native and hybrid local Mobile SDK apps on Android and iOS. It is not currently supported for hybrid remote apps using Visualforce.

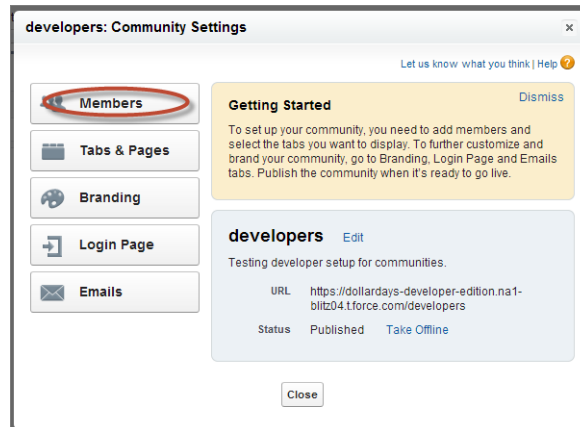
Set Up an API-Enabled Profile

If you're new to communities, start by enabling the community feature in your org. See [Enabling Salesforce Communities in Salesforce Help](#). When you're asked to create a domain name, be sure that it doesn't use SSL (<https://>).

To set up your community, see [Creating Communities in Salesforce Help](#). Note that you'll define a community URL based on the domain name you created when you enabled the community feature.

Next, configure one or more profiles with the API Enabled permissions. You can use these profiles to enable your Mobile SDK app for community members. For detailed instructions, follow the tutorial at [Example: Configure a Community 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, then edit your community at **Settings > Customize > Communities > Manage Communities**.
4. In *<your community>*: Community Settings, click **Members**.



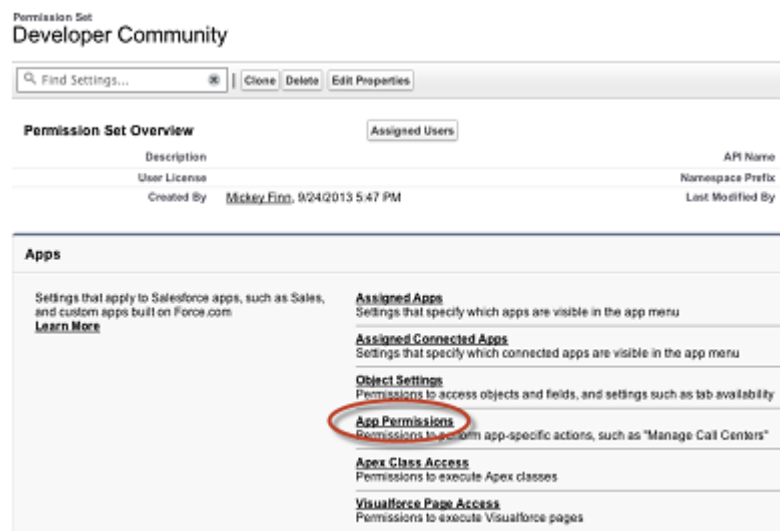
5. Add your API-enabled profile to **Selected Profiles**.

Users to whom these profiles are assigned now have API access. For an overview of profiles, see [User Profiles Overview](#) in Salesforce Help.

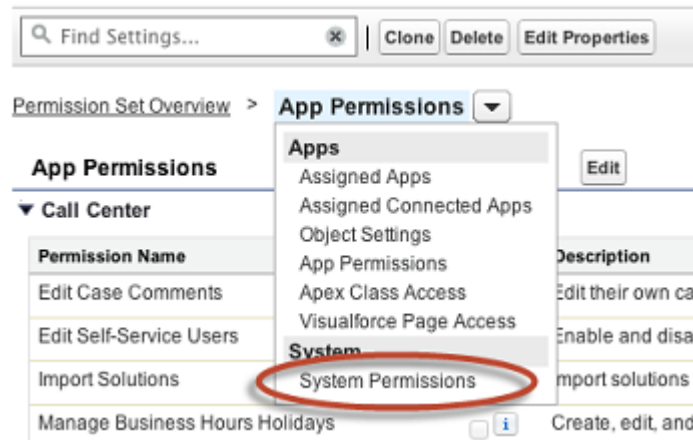
Set Up a Permission Set

Another way to enable mobile apps for your community is through a permission set.

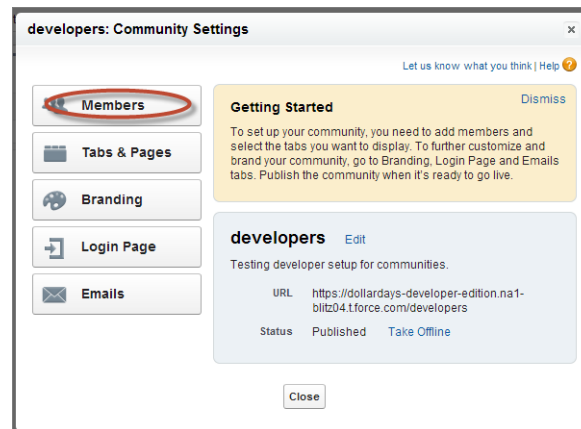
1. To add the API Enabled permission to an existing permission set, in Setup, click **Manage Users > Permissions Sets**, select the permission set, and skip to Step 6.
2. To create a permission set, in Setup, click **Administer > Manage Users > Permission Sets**.
3. Click **New**.
4. Give the Permission 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. Go to **Settings > Customize > Communities > Manage Communities** and click **Edit** next to your community name.
11. In *My Community*: Community Settings, 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 community 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 community login endpoint. The app's mobile platform determines how you configure this setting.

Android

In Android, login hosts are known as server connections. Prior to Mobile SDK v. 1.4, server connections for Android apps were hard-coded in the SalesforceSDK project. In v. 1.4 and later, the host list is defined in the `res/xml/servers.xml` file. The SalesforceSDK library project uses this file to define production and sandbox servers. You can add your servers to the

runtime list by creating your own `res/xml/servers.xml` file in your application project. The root XML element for this file 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.)

For example:

```
<?xml version="1.0" encoding="utf-8"?>
<servers>
  <server name="XYZ.com Login" url="https://<username>.cloudforce.com"/>
</servers>
```

iOS

For iOS apps, you set the Custom Host in your app’s iOS settings bundle. If you’ve configured this setting, it will be used as the default connection. Add the following key-value pair to your `<appname>-Info.plist` file:

```
<key>SFDCOAuthLoginHost</key>
<string>your_community_login_url_minus_the_https://_prefix</string>
```

It’s important to remove the HTTP prefix from your URL. For example, if your community login URL is `https://mycommunity-developer-edition.na15.force.com/fineapps`, your key-value pair would be:

```
<key>SFDCOAuthLoginHost</key>
<string>mycommunity-developer-edition.na15.force.com/fineapps</string>
```

Optionally, you can remove `Settings.bundle` from your class if you don’t want users to change the value.

Branding Your Community

Customize the look and feel of your community by adding your company logo, colors, and copyright. This ensures that your community matches your company’s branding and is instantly recognizable to your community members.

Available in: **Enterprise, Performance, Unlimited, and Developer** Editions

User Permissions Needed	
To create, customize, or publish a community:	“Create and Customize Communities”

1. From Setup, click **Customize > Communities > Manage Communities**, then click **Edit** next to the community name.
2. Click **Branding**.
3. Use the lookups to choose a header and footer for the community.

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. So, if you have a header .html file that is 70 KB and you want to use an .html file for the footer as well, it can only be 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. Click **Select Color Scheme** to select from predefined color schemes or click the text box next to the page section fields to select a color from the color picker.

Note that some of the selected colors impact your community login page and how your community looks in Salesforce1 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.
Page Background	Background color for all pages in your community, 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**.

Customizing Communities Login

Customize the look and feel of your community login page, from the logo and footer to login options for external users.



Available in: **Enterprise, Performance, Unlimited, and Developer** Editions

User Permissions Needed	
To create, customize, or publish a community:	“Create and Customize Communities”

The colors used on the login page are inherited from the community [branding color scheme](#). You can customize these other elements of the page.

1. From Setup, click **Customize > Communities > Manage Communities**, then click **Edit** next to the community name.
2. Click **Login Page**.
3. Upload a logo for the community login page header.
The file can be .gif, .jpg, or .png. The maximum file size is 100 KB. Images larger than 250 pixels wide or 125 pixels high aren't accepted. Uploading a logo automatically creates a Communities Shared Document Folder on the Documents tab and saves the logo there. Once created, you can't delete the folder.
The header logo displays at the top left of the login page. It is also used when you access the community in Salesforce1.
4. Enter custom text for the community login page footer, up to a maximum of 120 characters.
The footer displays at the bottom of the login page.
5. Choose the login options to make available to external users on the community login page.
External users are users with Community, Customer Portal, or partner portal licenses.

Login Options for External Users	What Displays on the Login Page
Username and password to log in to <i>Organization Name</i>	The option to log in using the username and password that the user was assigned for the community. This is the default login option.

Login Options for External Users	What Displays on the Login Page
<p>SAML for single sign-on</p> <p>This option is available only if your organization has successfully set up both of the following.</p> <ul style="list-style-type: none"> • SAML settings for single sign-on, which enables login to Salesforce using your corporate identity provider. Note that you must enter an Identity Provider Login URL. • A custom Salesforce domain name, which changes the application URLs for all of your pages, including login pages. Contact Support if you need to enable My Domain. <p>You can offer users multiple SAML single sign-on options if you Enable Multiple Configs from Setup, in Security Controls > Single Sign-On Settings. If you already had SAML enabled and you then enable multiple SAML configurations, your existing SAML configuration is automatically converted to work with multiple additional configurations.</p>	<p>The option to Log In with Single Sign-On using the user's SAML single sign-on identity.</p> <p>If you have enabled multiple SAML single sign-on options, each login button displays labeled with the SAML configuration's Name field.</p>
<p>External authentication providers</p> <p>These options are available if you enable them from Setup, in Security Controls > Auth. Providers.</p>	<p>The option to log in using credentials from an external service provider such as Facebook®, Janrain®, or Salesforce.</p>
<p>Self-registration</p> <p>Rather than relying solely on community administrators to add members, you can Allow external users to self-register and select a default profile. This profile is assigned to users who self-register.</p> <p> Note: You can only select portal profiles that are associated with the community.</p> <p>If a profile is selected as the default for users who self-register, and you remove it from the community, the Default profile for users that self register is automatically reset to None.</p> <p>When you create your first community, a default set of self-registration Visualforce pages and associated Apex controllers are created. You must specify in the default controller which account the self-registration process should assign users to. You can also specify a profile, but it will override the default selected when enabling self-registration. The self-registration feature won't work until you specify these details.</p> <p> Note: Keep in mind that each time a user self registers, they consume one of your Communities licenses. When setting up your self registration page, be sure to add some criteria to ensure the right people are signing up. Additionally, to prevent unauthorized form submissions, we recommend using a security mechanism, such as CAPTCHA or a hidden field, on your self registration page.</p>	<p>A Not a member? link that directs external users to the self-registration page.</p>

6. Click **Save**.

Your selected login options will be visible to all users on the login page. However, they're valid only for external users. Internal users who try to use these options will get a login error. They must use the link that directs employees to **Log in here** and log in with their Salesforce username and password.

Using External Authentication With Communities

You can use an external authentication provider, such as Facebook[®], to log community 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.

About External Authentication Providers

User Permissions Needed	
To view the settings:	"View Setup and Configuration"
To edit the settings:	"Customize Application"
	AND
	"Manage Auth. Providers"

You can enable users to log into your Salesforce organization using their login credentials from an external service provider such as Facebook[®] or Janrain[®]. You must do the following to successfully set up an authentication provider for single sign-on.

- Correctly configure the service provider website.
- Create a registration handler using Apex.
- Define the authentication provider in your organization.



Note: Users with profiles containing login IP range restrictions or organizations using session locking can't use authentication providers.

After your authentication provider is set up, the basic flow is the following.

1. The user tries to login to Salesforce using a third party identity.
2. The login request is redirected to the third party authentication provider.
3. The user performs the third party login and approves access.
4. The authentication provider redirects the user to Salesforce with credentials.
5. The user is signed into Salesforce.



Note: If a user has an existing Salesforce session, after authentication with the third party they are automatically redirected to the page where they can approve the link to their Salesforce account.

Defining Your Authentication Provider

We support the following providers:

- [Facebook](#)
- [Janrain](#)
- [Salesforce](#)

- [Any service provider who implements the OpenID Connect protocol](#)
- Microsoft Access Control Service

Adding Functionality to Your Authentication Provider

You can add functionality to your authentication provider by using additional request parameters.

- [Scope](#) – Customizes the permissions requested from the third party
- [Site](#) – Enables the provider to be used with a site
- [StartURL](#) – Sends the user to a specified location after authentication
- [Community](#) – Sends the user to a specific community after authentication
- “Authorization Endpoint” in the Salesforce Help – Sends the user to a specific endpoint for authentication (Salesforce authentication providers, only)

Creating an Apex Registration Handler

A registration handler class is required to use Authentication Providers for the single sign-on flow. The Apex registration handler class must implement the `Auth.RegistrationHandler` interface, which defines two methods. Salesforce invokes the appropriate method on callback, depending on whether the user has used this provider before or not. When you create the authentication provider, you can automatically create an Apex template class for testing purposes. For more information, see [RegistrationHandler](#) in the *Force.com Apex Code Developer's Guide*.

Using the Community URL Parameter

Send your user to a specific Community after authenticating.

Available in: **Professional, Enterprise, Performance, Unlimited, and Developer** Editions

User Permissions Needed	
To view the settings:	“View Setup and Configuration”
To edit the settings:	“Customize Application”
	AND
	“Manage Auth. Providers”

To direct your users to a specific community after authenticating, you need to specify a URL with the `community` request parameter. If you don't add the parameter, the user is sent to either `/home/home.jsp` (for a portal or standard application) or to the default sites page (for a site) after authentication completes.

For example, with a Single Sign-On Initialization URL, the user is sent to this location after being logged in. For an Existing User Linking URL, the “Continue to Salesforce” link on the confirmation page leads to this page.

The following is an example of a `community` parameter added to the Single Sign-On Initialization URL, where:

- `orgID` is your Auth. Provider ID
- `URLsuffix` is the value you specified when you defined the authentication provider

`https://login.salesforce.com/services/auth/sso/orgID/URLsuffix?community=https://acme.force.com/support`

Using the Scope Parameter

Customizes the permissions requested from the third party like Facebook or Janrain so that the returned access token has additional permissions.

Available in: **Professional, Enterprise, Performance, Unlimited, and Developer** Editions

User Permissions Needed	
To view the settings:	“View Setup and Configuration”
To edit the settings:	“Customize Application”
	AND
	“Manage Auth. Providers”

You can customize requests to a third party to receive access tokens with additional permissions. Then you use `Auth.AuthToken` methods to retrieve the access token that was granted so you can use those permissions with the third party.

The default scopes vary depending on the third party, but usually do not allow access to much more than basic user information. Every provider type (Open ID Connect, Facebook, Salesforce, and others), has a set of default scopes it sends along with the request to the authorization endpoint. For example, Salesforce’s default scope is `id`.

You can send scopes in a space-delimited string. The space-delimited string of requested scopes is sent as-is to the third party, and overrides the default permissions requested by authentication providers.


Janrain does not use this parameter; additional permissions must be configured within Janrain.

The following is an example of a `scope` parameter requesting the Salesforce scopes `api` and `web`, added to the `Single Sign-On Initialization URL`, where:

- `orgID` is your Auth. Provider ID
- `URLsuffix` is the value you specified when you defined the authentication provider

```
https://login.salesforce.com/services/auth/sso/orgID/URLsuffix?scope=id%20api%20web
```

Valid scopes vary depending on the third party; refer to your individual third-party documentation. For example, Salesforce scopes are:

Value	Description
<code>api</code>	Allows access to the current, logged-in user’s account using APIs, such as REST API and Bulk API. This value also includes <code>chatter_api</code> , which allows access to Chatter REST API resources.
<code>chatter_api</code>	Allows access to Chatter REST API resources only.
<code>custom_permissions</code>	Allows access to the custom permissions in an organization associated with the connected app, and shows whether the current user has each permission enabled.
 Note: Custom permissions are currently available as a Developer Preview.	

Value	Description
full	Allows access to all data accessible by the logged-in user, and encompasses all other scopes. <code>full</code> does not return a refresh token. You must explicitly request the <code>refresh_token</code> scope to get a refresh token.
id	Allows access to the identity URL service. You can request <code>profile</code> , <code>email</code> , <code>address</code> , or <code>phone</code> , individually to get the same result as using <code>id</code> ; they are all synonymous.
openid	Allows access to the current, logged in user's unique identifier for OpenID Connect apps. The <code>openid</code> scope can be used in the OAuth 2.0 user-agent flow and the OAuth 2.0 Web server authentication flow to get back a signed ID token conforming to the OpenID Connect specifications in addition to the access token.
refresh_token	Allows a refresh token to be returned if you are eligible to receive one. This lets the app interact with the user's data while the user is offline, and is synonymous with requesting <code>offline_access</code> .
visualforce	Allows access to Visualforce pages.
web	Allows the ability to use the <code>access_token</code> on the Web. This also includes <code>visualforce</code> , allowing access to Visualforce pages.

Configuring a Facebook Authentication Provider

User Permissions Needed	
To view the settings:	"View Setup and Configuration"
To edit the settings:	"Customize Application"
	AND
	"Manage Auth. Providers"

To use Facebook as an authentication provider:

1. [Set up](#) a Facebook application, making Salesforce the application domain.
2. [Define](#) a Facebook authentication provider in your Salesforce organization.
3. [Update](#) your Facebook application to use the `Callback URL` generated by Salesforce as the `Facebook Website Site URL`.
4. [Test](#) the connection.

Setting up a Facebook Application

Before you can configure Facebook for your Salesforce organization, you must set up an application in Facebook:

1. Go to the [Facebook website](#) and create a new application.
2. Modify the application settings and set the Application Domain to Salesforce.
3. Note the Application ID and the Application Secret.

Defining a Facebook Provider in your Salesforce Organization

You need the Facebook Application ID and Application Secret to set up a Facebook provider in your Salesforce organization.

1. From Setup, click **Security Controls > Auth. Providers**.

2. Click **New**.
3. Select Facebook for the `Provider Type`.
4. Enter a Name for the provider.
5. Enter the `URL Suffix`. This is used in the client configuration URLs. For example, if the URL suffix of your provider is “MyFacebookProvider”, your single sign-on URL is similar to:
`https://login.salesforce.com/auth/sso/00Dx000000000001/MyFacebookProvider`.
6. Use the Application ID from Facebook for the `Consumer Key` field.
7. Use the Application Secret from Facebook for the `Consumer Secret` field.
8. Optionally, set the following fields.

- a. `Default Scopes` to send along with the request to the authorization endpoint. Otherwise, the hardcoded defaults for the provider type are used (see [Facebook’s developer documentation](#) for these defaults).

For more information, see [Using the Scope Parameter](#)

- b. `Custom Error URL` for the provider to use to report any errors.
- c. Select an already existing Apex class as the `Registration Handler` class or click `Automatically create a registration handler template` to create an Apex class template for the registration handler. You must edit this class and modify the default content before using it.



Note: You must specify a registration handler class for Salesforce to generate the `Single Sign-On Initialization URL`.

- d. Select the user that runs the Apex handler class for **Execute Registration As**. The user must have “Manage Users” permission. A user is required if you selected a registration handler class or are automatically creating one.
- e. To use a portal with your provider, select the portal from the Portal drop-down list.

9. Click **Save**.

Be sure to note the generated `Auth. Provider Id` value. You must use it with the `Auth.AuthToken` Apex class.

Several client configuration URLs are generated after defining the authentication provider:

- **Test-Only Initialization URL:** Administrators use this URL to ensure the third-party provider is set up correctly. The administrator opens this URL in a browser, signs in to the third party, and is redirected back to Salesforce with a map of attributes.
- **Single Sign-On Initialization URL:** Use this URL to perform single sign-on into Salesforce from a third party (using third-party credentials). The end user opens this URL in a browser, and signs in to the third party. This then either creates a new user for them, or updates an existing user, and then signs them into Salesforce as that user.
- **Existing User Linking URL:** Use this URL to link existing Salesforce users to a third-party account. The end 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 does not provide for future single sign-on functionality.
- **Callback URL:** Use the callback URL for the endpoint that the authentication provider calls back to for configuration. The authentication provider has to redirect to the `Callback URL` with information for each of the above client configuration URLs.

The client configuration URLs support additional request parameters that enable you to direct users to log into specific sites, obtain customized permissions from the third party, or go to a specific location after authenticating.

Updating Your Facebook Application

After defining the Facebook authentication provider in your Salesforce organization, go back to Facebook and update your application to use the `Callback URL` as the `Facebook Website Site URL`.

Testing the Single Sign-On Connection

In a browser, open the `Test-Only Initialization URL` on the Auth. Provider detail page. It should redirect you to Facebook and ask you to sign in. Upon doing so, you are asked to authorize your application. After you authorize, you are redirected back to Salesforce.

Configuring a Salesforce Authentication Provider

User Permissions Needed	
To view the settings:	“View Setup and Configuration”
To edit the settings:	“Customize Application”
	AND
	“Manage Auth. Providers”

You can use a connected app as an authentication provider. You must complete these steps:

1. [Define a Connected App](#).
2. [Define the Salesforce authentication provider in your organization](#).
3. [Test the connection](#).

Defining a Connected App

Before you can configure a Salesforce provider for your Salesforce organization, you must define a connected app that uses single sign-on. Define connected apps under Setup, in **Create > Apps**.

After you finish defining a connected app, save the values from the `Consumer Key` and `Consumer Secret` fields.

Defining the Salesforce Authentication Provider in your Organization

You need the values from the `Consumer Key` and `Consumer Secret` fields of the connected app definition to set up the authentication provider in your organization.

1. From Setup, click **Security Controls > Auth. Providers**.
2. Click **New**.
3. Select Salesforce for the `Provider Type`.
4. Enter a `Name` for the provider.
5. Enter the `URL Suffix`. This is used in the client configuration URLs. For example, if the URL suffix of your provider is “MySFDCProvider”, your single sign-on URL is similar to
`https://login.salesforce.com/auth/sso/00Dx000000000001/MySFDCProvider`.
6. Paste the value of `Consumer Key` from the connected app definition into the `Consumer Key` field.
7. Paste the value of `Consumer Secret` from the connected app definition into the `Consumer Secret` field.
8. Optionally, set the following fields.
 - a. `Authorize Endpoint URL` to specify an OAuth authorization URL.
 For the `Authorize Endpoint URL`, the host name can include a sandbox or custom domain name (created using My Domain), but the URL must end in `.salesforce.com`, and the path must end in `/services/oauth2/authorize`. For example
`https://test.salesforce.com/services/oauth2/authorize`.
 - b. `Token Endpoint URL` to specify an OAuth token URL.

For the `Token Endpoint URL`, the host name can include a sandbox or custom domain name (created using My Domain), but the URL must end in `.salesforce.com`, and the path must end in `/services/oauth2/token`. For example `https://test.salesforce.com/services/oauth2/token`.

- c. `Default Scopes` to send along with the request to the authorization endpoint. Otherwise, the hardcoded default is used.

For more information, see [Using the Scope Parameter](#).

- d. `Custom Error URL` for the provider to use to report any errors.

9. Select an already existing Apex class as the `Registration Handler` class or click `Automatically create a registration handler template` to create the Apex class template for the registration handler. You must edit this template class to modify the default content before using it.



Note: You must specify a registration handler class for Salesforce to generate the `Single Sign-On Initialization URL`.

10. Select the user that runs the Apex handler class for `Execute Registration As`. The user must have “Manage Users” permission. A user is required if you selected a registration handler class or are automatically creating one.
11. To use a portal with your provider, select the portal from the `Portal` drop-down list.
12. Click `Save`.

Note the value of the `Client Configuration URLs`. You need the `Callback URL` to complete the last step, and you use the `Test-Only Initialization URL` to check your configuration. Also be sure to note the `Auth. Provider Id` value because you must use it with the `Auth.AuthToken` Apex class.

13. Return to the connected app definition you created above (under `Setup`, in **Create > Apps**, click on the connected app name) and paste the value of `Callback URL` from the authentication provider into the `Callback URL` field.

Several client configuration URLs are generated after defining the authentication provider:

- `Test-Only Initialization URL`: Administrators use this URL to ensure the third-party provider is set up correctly. The administrator opens this URL in a browser, signs in to the third party, and is redirected back to Salesforce with a map of attributes.
- `Single Sign-On Initialization URL`: Use this URL to perform single sign-on into Salesforce from a third party (using third-party credentials). The end user opens this URL in a browser, and signs in to the third party. This then either creates a new user for them, or updates an existing user, and then signs them into Salesforce as that user.
- `Existing User Linking URL`: Use this URL to link existing Salesforce users to a third-party account. The end 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 does not provide for future single sign-on functionality.
- `Callback URL`: Use the callback URL for the endpoint that the authentication provider calls back to for configuration. The authentication provider has to redirect to the `Callback URL` with information for each of the above client configuration URLs.

The client configuration URLs support additional request parameters that enable you to direct users to log into specific sites, obtain customized permissions from the third party, or go to a specific location after authenticating.

Testing the Single Sign-On Connection

In a browser, open the `Test-Only Initialization URL` on the `Auth. Provider` detail page. Both the authorizing organization and target organization must be in the same environment, such as production or a sandbox.

Configuring an OpenID Connect Authentication Provider

You can use any third-party Web application that implements the server side of the OpenID Connect protocol, such as Amazon, Google, and PayPal, as an authentication provider.

User Permissions Needed	
To view the settings:	“View Setup and Configuration”
To edit the settings:	“Customize Application”
	AND
	“Manage Auth. Providers”

You must complete these steps to configure an OpenID authentication provider:

1. [Register](#) your application, making Salesforce the application domain.
2. [Define](#) an OpenID Connect authentication provider in your Salesforce organization.
3. [Update](#) your application to use the `Callback URL` generated by Salesforce as the callback URL.
4. [Test](#) the connection.

Registering an OpenID Connect Application

Before you can configure a Web application for your Salesforce organization, you must register it with your service provider. The process varies depending on the service provider. For example, to register a Google app, [Create an OAuth 2.0 Client ID](#).

1. Register your application on your service provider’s website.
2. Modify the application settings and set the application domain (or `Home Page URL`) to Salesforce.
3. Note the Client ID and Client Secret, as well as the Authorize Endpoint URL, Token Endpoint URL, and User Info Endpoint URL, which should be available in the provider’s documentation. Here are some common OpenID Connect service providers:
 - [Amazon](#)
 - [Google](#)
 - [PayPal](#)

Defining an OpenID Connect Provider in Your Salesforce Organization

You need some information from your provider (the Client ID and Client Secret, as well as the Authorize Endpoint URL, Token Endpoint URL, and User Info Endpoint URL) to configure your application in your Salesforce organization.

1. From Setup, click **Security Controls > Auth. Providers**.
2. Click **New**.
3. Select OpenID Connect for the `Provider Type`.
4. Enter a `Name` for the provider.
5. Enter the `URL Suffix`. This is used in the client configuration URLs. For example, if the URL suffix of your provider is “MyOpenIDConnectProvider,” your single sign-on URL is similar to:
`https://login.salesforce.com/auth/sso/00Dx000000000001/MyOpenIDConnectProvider.`
6. Use the Client ID from your provider for the `Consumer Key` field.
7. Use the Client Secret from your provider for the `Consumer Secret` field.
8. Enter the base URL from your provider for the `Authorize Endpoint URL`.



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`. In this specific case, the additional `approval_prompt` parameter is necessary to ask the user to accept the refresh action, so Google will continue to provide refresh tokens after the first one.

9. Enter the `Token Endpoint URL` from your provider.

10. Optionally, set the following fields.

- a. `User Info Endpoint URL` from your provider.
- b. `Token Issuer`. This value identifies the source of the authentication token in the form `https://URL`.
- c. `Default Scopes` to send along with the request to the authorization endpoint. Otherwise, the hardcoded defaults for the provider type are used (see the [OpenID Connect developer documentation](#) for these defaults).

For more information, see [Using the Scope Parameter](#)

11. You can select `Send access token in header` to have the token sent in a header instead of a query string.

12. Optionally, set the following fields.

- a. `Custom Error URL` for the provider to use to report any errors.
- b. Select an existing Apex class as the `Registration Handler` class or click `Automatically create a registration handler template` to create an Apex class template for the registration handler. You must edit this class and modify the default content before using it.



Note: You must specify a registration handler class for Salesforce to generate the `Single Sign-On Initialization URL`.

- c. Select the user that runs the Apex handler class for **Execute Registration As**. The user must have the “Manage Users” permission. A user is required if you selected a registration handler class or are automatically creating one.
- d. To use a portal with your provider, select the portal from the `Portal` drop-down list.

13. Click **Save**.

Be sure to note the generated `Auth. Provider Id` value. You must use it with the `Auth.AuthToken` Apex class.

Several client configuration URLs are generated after defining the authentication provider:

- **Test-Only Initialization URL:** Administrators use this URL to ensure the third-party provider is set up correctly. The administrator opens this URL in a browser, signs in to the third party, and is redirected back to Salesforce with a map of attributes.
- **Single Sign-On Initialization URL:** Use this URL to perform single sign-on into Salesforce from a third party (using third-party credentials). The end user opens this URL in a browser, and signs in to the third party. This then either creates a new user for them, or updates an existing user, and then signs them into Salesforce as that user.
- **Existing User Linking URL:** Use this URL to link existing Salesforce users to a third-party account. The end 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 does not provide for future single sign-on functionality.
- **Callback URL:** Use the callback URL for the endpoint that the authentication provider calls back to for configuration. The authentication provider has to redirect to the `Callback URL` with information for each of the above client configuration URLs.

The client configuration URLs support additional request parameters that enable you to direct users to log into specific sites, obtain customized permissions from the third party, or go to a specific location after authenticating.

Updating Your OpenID Connect Application

After defining the authentication provider in your Salesforce organization, go back to your provider and update your application's `Callback URL` (also called the `Authorized Redirect URI` for Google applications and `Return URL` for PayPal).

Testing the Single Sign-On Connection

In a browser, open the `Test-Only Initialization URL` on the Auth. Provider detail page. It should redirect you to your provider's service and ask you to sign in. Upon doing so, you're asked to authorize your application. After you authorize, you're redirected back to Salesforce.

Example: Configure a Community For Mobile SDK App Access

Configuring your community 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 community users for mobile access, sequence and protocol affect your success. For example, if you create a user that's not associated with a contact, that user won't be able to 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.
- On iOS devices, when you create a Custom Host for your app in Settings, remove the `http[s]://` prefix. The iOS core appends the prefix at runtime, which could result in an invalid address if you explicitly include it.

1. [Add Permissions to a Profile](#)
2. [Create a Community](#)
3. [Add the API User Profile To Your Community](#)
4. [Create a New Contact and User](#)
5. [Test Your New Community Login](#)

Add Permissions to a Profile

Create a profile that has API Enabled and Enable Chatter permissions.

1. Go to **Setup > Manage Users > 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 community 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 a Community

Create a community and a community login URL.

The following steps are fully documented at [Enabling Salesforce Communities](#) and [Creating Communities](#) in Salesforce Help.

1. In **Setup**, go to **Customize > Communities**.
2. If you don't see a **Manage Communities** options:
 - a. Click **Settings**.
 - b. Under Enable communities, select **Enable communities**.
 - c. Under Select a domain name, enter a unique 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. Go to **Setup > Customize > Communities > Manage Communities**.
4. Click **New Community**.
5. Name the new community `FineApps Users` and enter a description.
6. For **URL**, type `customers` in the suffix edit box.

The full URL shown, including your suffix, becomes the new URL for your community.
7. Click **Create**, then click **Edit**.

Add the API User Profile To Your Community

Add the API User profile to your community setup on the Members page.

1. Click **Members**.
2. For Search, select **All**.
3. Select **FineApps API User** in the Available Profiles list, 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, 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. Go to **Manage Users > Users** and 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**, 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, then click the link in the email to create a password. Set the password to "mobile333".

Test Your New Community Login

Test your community setup by logging into your Mobile SDK native or hybrid local app as your new contact.

To log into your mobile app through your community, configure the settings in your Mobile SDK app to recognize your community login URL that ends with `/fineapps`.

1. For Android:
 - a. Open your Android project in Eclipse.
 - 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 new 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 community login URL:

```
<?xml version="1.0" encoding="utf-8"?>
<servers>
  <server name="Community Login"
    url="https://fineapps-developer-edition.<instance>.force.com/fineapps">
  </server>
</servers>
```

- e. Save the file as `servers.xml`.

2. For iOS:
 - a. Open your iOS project in Xcode.
 - b. Using the Project Navigator, open **Supporting Files** > **<appname>-Info.plist**.
 - c. Change the **SFDCOAuthLoginHost** value to your community login URL minus the `https://` prefix. For example:

```
fineapps-developer-edition.<instance>.force.com/fineapps
```

- d. On your iOS simulator or device, go to **Settings** > **<your_app_name>**.
 - e. Click **Login Host** and select **Custom Host**.

- f. Click **Back**.
- g. Edit **Custom Host**, setting it to the `SFDCOAuthLoginHost` value you specified in the `<appname>-Info.plist` file.
3. Start your app on your device, simulator, or emulator, and log in with username `jimparker@fineapps.com` and password `mobiletest1234`.



Note: If you leave your mobile app at the login screen for an extended time without logging in, you might get an “insufficient privileges” error when you try to log in. If this happens, close and reopen the app, then log in immediately.

Example: Configure a Community For Facebook Authentication

You can extend the reach of your community by configuring an external authentication provider to handle community 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 community.

If your community 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 community setup.

Create a Facebook App

To enable community 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 Community 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, go to **Security Controls > 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 a community 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 community 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 `login.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 community.
8. Click **Save**.
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 community.

1. In Setup, go to **Develop > Apex Classes**.
2. Click **Edit** next to your Auth. Provider class. The default class name starts with “AutocreatedRegHandlerxxxxx...”
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 community.



Note: If you want your community to be accessible only to existing community members, implement a filter to recognize every valid user in your community. 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 community 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 Salesforce Community

For the final step, configure the community to use your Salesforce Auth. Provider for logins.

1. In Setup, go to **Customize > Communities > Manage Communities**.
2. Click **Edit** next to your community name.
3. Click **Login Page**.
4. Under Options for External Users, select your new Auth. Provider.
5. Click **Save**.

You’re done! Now, when you log into your mobile app using your community 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://login.salesforce.com/services/auth/sso/orgID/
URLsuffix?community=<community_login_url>
```

For example:

```
https://login.salesforce.com/services/auth/sso/00Da0000000TPNEAA4/
FB_Community_Login?community=
https://mobilesdk-developer-edition.server_instance.force.com/fineapps
```

To form the Existing User Linking URL, replace `sso` with `link`:

```
https://login.salesforce.com/services/auth/link/00Da0000000TPNEAA4/  
FB_Community_Login?community=  
https://mobilesdk-developer-edition.server_instance.force.com/fineapps
```

Chapter 10

Authentication, Security, and Identity in Mobile Apps

In this chapter ...

- [OAuth Terminology](#)
- [OAuth2 Authentication Flow](#)
- [Connected Apps](#)
- [Portal Authentication Using OAuth 2.0 and Force.com Sites](#)

Secure authentication is essential for enterprise applications running on mobile devices. OAuth2 is the industry-standard protocol that allows secure authentication for access to a user's data, without handing out the username and password. It is often described as the valet key of software access: a valet key only allows access to certain features of your car; you cannot open the trunk or glove compartment using a valet key.

Mobile app developers can quickly and easily embed the Salesforce OAuth2 implementation. The implementation uses an HTML view to collect the username and password, which are then sent to the server. A session token is returned and securely stored on the device for future interactions.

A Salesforce *connected app* is the primary means by which a mobile device 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 be restricted to certain users, can set or relax an IP range, and so forth.

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. Replaces remote access application.

Consumer Key

A value used by the consumer to identify itself to Salesforce. Referred to as `client_id`.

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” in the Salesforce Help. Remote access applications have been deprecated in favor of connected apps.

OAuth2 Authentication Flow

The authentication flow depends on the state of authentication on the device.

First Time Authentication Flow

1. User opens a mobile application.
2. An authentication dialog/window/overlay appears.
3. User enters username and password.
4. App receives session ID.
5. User grants access to the app.
6. App starts.

Ongoing Authentication

1. User opens a mobile application.
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. App starts.

PIN Authentication (Optional)

1. User opens a mobile application after not using it for some time.
2. If the elapsed time exceeds the configured PIN timeout value, a passcode entry screen appears. User enters the PIN.



Note: PIN protection is a function of the mobile policy and is used only when it's enabled in the Salesforce connected app definition. It can be shown whether you are online or offline, if enough time has elapsed since you last used the application. See [About PIN Security](#).

3. App re-uses existing session ID.
4. App starts.

OAuth 2.0 User-Agent Flow

The user-agent authentication flow is used by client applications residing on the user's mobile device. The authentication is based on the user-agent's same-origin policy.

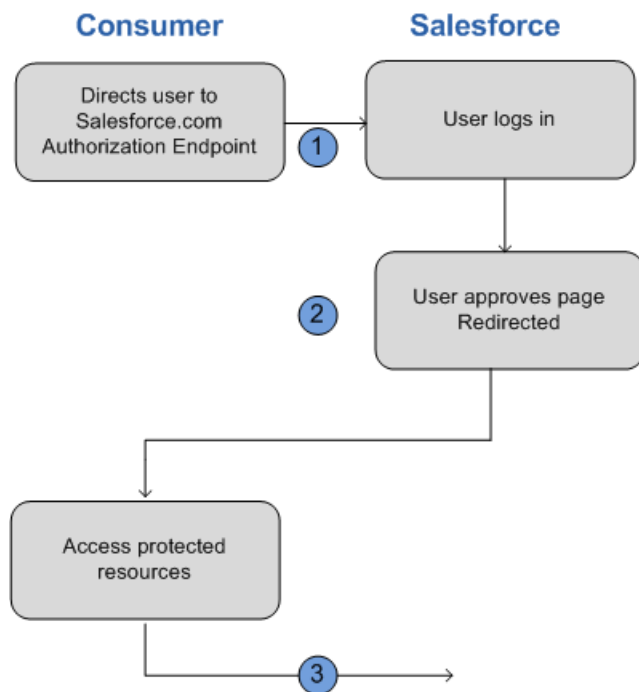
In the user-agent flow, the client application receives the access token in the form of an HTTP redirection. The client application requests the authorization server to redirect the user-agent to another web server or local resource accessible to the user-agent, which is capable of extracting the access token from the response and passing it to the client application. Note that the token response is provided as a hash (#) fragment on the URL. This is for security, and prevents the token from being passed to the server, as well as to other servers in referral headers.

This user-agent authentication flow doesn't utilize the client secret since the client executables reside on the end-user's computer or device, which makes the client secret accessible and exploitable.



Warning: Because the access token is encoded into the redirection URI, it might be exposed to the end-user and other applications residing on the computer or device.

If you are authenticating using JavaScript, call `window.location.replace()` ; to remove the callback from the browser's history.



1. The client application directs the user to Salesforce to authenticate and authorize the application.
2. The user must always approve access for this authentication flow. After approving access, the application receives the callback from Salesforce.

After obtaining an access token, the consumer can use the access token to access data on the end-user's behalf and receive a refresh token. Refresh tokens let the consumer get a new access token if the access token becomes invalid for any reason.

OAuth 2.0 Refresh Token Flow

After the consumer has been authorized for access, they can use a refresh token to get a new access token (session ID). This is only done after the consumer already has received a refresh token using either the Web server or user-agent flow. It is up to the consumer to determine when an access token is no longer valid, and when to apply for a new one. Bearer flows can only be used after the consumer has received a refresh token.

The following are the steps for the refresh token authentication flow. More detail about each step follows:

1. The consumer uses the existing refresh token to request a new access token.
2. After the request is verified, Salesforce sends a response to the client.



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.


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 scopes client apps, such as Mobile SDK apps, can request. For most native Mobile SDK apps, `refresh_token` and `api` are sufficient.
- **Client side**—Define scope requests in your Mobile SDK app. Client scope requests must be a subset of the connected app's scope permissions.

Server Side Configuration

The `scope` parameter enables you to fine-tune what the client application can access in a Salesforce organization. The valid values for `scope` are:

Value	Description
<code>api</code>	Allows access to the current, logged-in user's account using APIs, such as REST API and Bulk API. This value also includes <code>chatter_api</code> , which allows access to Chatter REST API resources.
<code>chatter_api</code>	Allows access to Chatter REST API resources only.
<code>custom_permissions</code>	Allows access to the custom permissions in an organization associated with the connected app, and shows whether the current user has each permission enabled.
 Note: Custom permissions are currently available as a Developer Preview.	

Value	Description
full	Allows access to all data accessible by the logged-in user, and encompasses all other scopes. <code>full</code> does not return a refresh token. You must explicitly request the <code>refresh_token</code> scope to get a refresh token.
id	Allows access to the identity URL service. You can request <code>profile</code> , <code>email</code> , <code>address</code> , or <code>phone</code> , individually to get the same result as using <code>id</code> ; they are all synonymous.
openid	Allows access to the current, logged in user's unique identifier for OpenID Connect apps. The <code>openid</code> scope can be used in the OAuth 2.0 user-agent flow and the OAuth 2.0 Web server authentication flow to get back a signed ID token conforming to the OpenID Connect specifications in addition to the access token.
refresh_token	Allows a refresh token to be returned if you are eligible to receive one. This lets the app interact with the user's data while the user is offline, and is synonymous with requesting <code>offline_access</code> .
visualforce	Allows access to Visualforce pages.
web	Allows the ability to use the <code>access_token</code> on the Web. This also includes <code>visualforce</code> , allowing access to Visualforce pages.



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 request.
api	Include in your request if you're making any Salesforce REST API calls (applies to most apps).
web	Include in your request if your app accesses pages defined in a Salesforce org (for hybrid apps, as well as native apps that load Salesforce-based Web pages.)
full	Include if you wish to request all permissions. (Mobile SDK implicitly requests <code>refresh_token</code> for you.)
chatter_api	Include in your request if your app calls Chatter REST APIs.
id	(Not needed)
visualforce	Use Web instead.

Using Identity URLs

In addition to the access token, an identity URL is also returned as part of a token response, in the `id` scope parameter.

The identity URL is both a string that uniquely identifies a user, as well as a RESTful API that can be used to query (with a valid access token) for additional information about the user. Salesforce returns basic personalization information about the user, as well as important endpoints that the client can talk to, such as photos for the user, and API endpoints it can access.

The format of the URL is: `https://login.salesforce.com/id/orgID/userID`, where *orgID* is the ID of the Salesforce organization that the user belongs to, and *userID* is the Salesforce user ID.



Note: For a sandbox, `login.salesforce.com` is replaced with `test.salesforce.com`.

The URL must always be HTTPS.

Identity URL Parameters

The following parameters can be used with the access token and identity URL. The access token can be used in an authorization request header or in a request with the `oauth_token` parameter.

Parameter	Description
Access token	See “Using the Access Token” in the Salesforce Help.
Format	<p>This parameter is optional. Specify the format of the returned output. Valid values are:</p> <ul style="list-style-type: none"> <code>urlencoded</code> <code>json</code> <code>xml</code> <p>Instead of using the <code>format</code> parameter, the client can also specify the returned format in an accept-request header using one of the following:</p> <ul style="list-style-type: none"> <code>Accept: application/json</code> <code>Accept: application/xml</code> <code>Accept: application/x-www-form-urlencoded</code> <p>Note the following:</p> <ul style="list-style-type: none"> Wildcard accept headers are allowed. <code>/*/*</code> is accepted and returns JSON. A list of values is also accepted and is checked left-to-right. For example: <code>application/xml,application/json,application/html,/*/*</code> returns XML. The <code>format</code> parameter takes precedence over the accept request header.
Version	This parameter is optional. Specify a SOAP API version number, or the literal string, <code>latest</code> . If this value isn’t specified, the returned API URLs contains the literal value <code>{version}</code> , in place of the version number, for the client to do string replacement. If the value is specified as <code>latest</code> , the most recent API version is used.
PrettyPrint	This parameter is optional, and is only accepted in a header, not as a URL parameter. Specify the output to be better formatted. For example, use the following in a header: <code>X-PrettyPrint:1</code> . If this value isn’t specified, the returned XML or JSON is optimized for size rather than readability.
Callback	<p>This parameter is optional. Specify a valid JavaScript function name. This parameter is only used when the format is specified as JSON. The output is wrapped in this function name (JSONP.) For example, if a request to <code>https://server/id/orgid/userid/</code> returns <code>{"foo":"bar"}</code>, a request to <code>https://server/id/orgid/userid/?callback=baz</code> returns <code>baz({"foo":"bar"});</code>.</p>

Identity URL Response

After making a valid request, a **302 redirect** to an instance URL is returned. That subsequent request returns the following information in JSON format:

- `id`—The identity URL (the same URL that was queried)
- `asserted_user`—A boolean value, indicating whether the specified access token used was issued for this identity
- `user_id`—The Salesforce user ID
- `username`—The Salesforce username
- `organization_id`—The Salesforce organization ID
- `nick_name`—The community nickname of the queried user
- `display_name`—The display name (full name) of the queried user
- `email`—The email address of the queried user
- `email_verified`—Indicates whether the organization has email verification enabled (`true`), or not (`false`).
- `first_name`—The first name of the user
- `last_name`—The last name of the user
- `timezone`—The time zone in the user's settings
- `photos`—A map of URLs to the user's profile pictures



Note: Accessing these URLs requires passing an access token. See “Using the Access Token” in the Salesforce Help.

- ◊ `picture`
- ◊ `thumbnail`

- `addr_street`—The street specified in the address of the user's settings
- `addr_city`—The city specified in the address of the user's settings
- `addr_state`—The state specified in the address of the user's settings
- `addr_country`—The country specified in the address of the user's settings
- `addr_zip`—The zip or postal code specified in the address of the user's settings
- `mobile_phone`—The mobile phone number in the user's settings
- `mobile_phone_verified`—The user confirmed this is a valid mobile phone number. See the Mobile User field description.
- `status`—The 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`: the body of the post

- `urls`—A map containing various API endpoints that can be used with the specified user



Note: Accessing the REST endpoints requires passing an access token. See “Using the Access Token” in the Salesforce Help.

- ◊ `enterprise` (SOAP)
- ◊ `metadata` (SOAP)
- ◊ `partner` (SOAP)
- ◊ `rest` (REST)
- ◊ `subjects` (REST)
- ◊ `search` (REST)
- ◊ `query` (REST)
- ◊ `recent` (REST)

- ◊ profile
 - ◊ feeds (Chatter)
 - ◊ feed-items (Chatter)
 - ◊ groups (Chatter)
 - ◊ users (Chatter)
 - ◊ custom_domain—This value is omitted if the organization doesn't have a custom domain configured and propagated
- active—A boolean specifying whether the queried user is active
 - user_type—The type of the queried user
 - language—The queried user's language
 - locale—The queried user's locale
 - utcOffset—The offset from UTC of the timezone of the queried user, in milliseconds
 - last_modified_date—xsd datetime format of last modification of the user, for example, 2010-06-28T20:54:09.000Z
 - is_app_installed—The value is true when the connected app is installed in the org of the current user and the access token for the user was created using an OAuth flow. If the connected app is not installed, the property does not exist (instead of being false). When parsing the response, check both for the existence and value of this property.
 - mobile_policy—Specific values for managing mobile connected apps. These values are only available when the connected app is installed in the organization of the current user and the app has a defined session timeout value and a PIN (Personal Identification Number) length value.
 - ◊ screen_lock—The length of time to wait to lock the screen after inactivity
 - ◊ pin_length—The length of the identification number required to gain access to the mobile app
 - push_service_type—This response value is set to apple if the connected app is registered with Apple Push Notification Service (APNS) for iOS push notifications or 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 an organization associated with the connected app. If the connected app is not installed in the organization, or has no associated custom permissions, the response does not contain a custom_permissions map. The following shows an example request.

```
http://login.salesforce.com/services/oauth2/authorize?response_type=token&client_id=3MVG91KcP0NINVBKV6EgVJiF.snSDwh6_2wSS7BrOhHGEJkC_&redirect_uri=http://www.example.org/qa/security/oauth/useragent_flow_callback.jsp&scope=api%20id%20custom_permissions
```

The following shows the JSON block in the identity URL response.

```
"custom_permissions":
{
  "Email.View":true,
  "Email.Create":false,
  "Email.Delete":false
}
```



Note: Custom permissions are currently available as a Developer Preview.

The following is a response in XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<user xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<id>http://na1.salesforce.com/id/00Dx0000001T0zk/005x0000001S2b9</id>
<asserted_user>true</asserted_user>
<user_id>005x0000001S2b9</user_id>
```

```

<organization_id>00Dx0000001T0zk</organization_id>
<nick_name>admin1.2777578168398293E12foofoofoofoo</nick_name>
<display_name>Alan Van</display_name>
<email>admin@2060747062579699.com</email>
<status>
  <created_date xsi:nil="true"/>
  <body xsi:nil="true"/>
</status>
<photos>
  <picture>http://na1.salesforce.com/profilephoto/005/F</picture>
  <thumbnail>http://na1.salesforce.com/profilephoto/005/T</thumbnail>
</photos>
<urls>
  <enterprise>http://na1.salesforce.com/services/Soap/c/{version}/00Dx0000001T0zk
</enterprise>
  <metadata>http://na1.salesforce.com/services/Soap/m/{version}/00Dx0000001T0zk
</metadata>
  <partner>http://na1.salesforce.com/services/Soap/u/{version}/00Dx0000001T0zk
</partner>
  <rest>http://na1.salesforce.com/services/data/v{version}/
</rest>
  <subjects>http://na1.salesforce.com/services/data/v{version}/subjects/
</subjects>
  <search>http://na1.salesforce.com/services/data/v{version}/search/
</search>
  <query>http://na1.salesforce.com/services/data/v{version}/query/
</query>
  <profile>http://na1.salesforce.com/005x0000001S2b9
</profile>
</urls>
<active>true</active>
<user_type>STANDARD</user_type>
<language>en_US</language>
<locale>en_US</locale>
<utcOffset>-28800000</utcOffset>
<last_modified_date>2010-06-28T20:54:09.000Z</last_modified_date>
</user>

```

The following is a response in JSON format:

```

{"id":"http://na1.salesforce.com/id/00Dx0000001T0zk/005x0000001S2b9",
"asserted_user":true,
"user_id":"005x0000001S2b9",
"organization_id":"00Dx0000001T0zk",
"nick_name":"admin1.2777578168398293E12foofoofoofoo",
"display_name":"Alan Van",
"email":"admin@2060747062579699.com",
"status":{"created_date":null,"body":null},
"photos":{"picture":"http://na1.salesforce.com/profilephoto/005/F",
"thumbnail":"http://na1.salesforce.com/profilephoto/005/T"},
"urls":
  {"enterprise":"http://na1.salesforce.com/services/Soap/c/{version}/00Dx0000001T0zk",
"metadata":"http://na1.salesforce.com/services/Soap/m/{version}/00Dx0000001T0zk",
"partner":"http://na1.salesforce.com/services/Soap/u/{version}/00Dx0000001T0zk",
"rest":"http://na1.salesforce.com/services/data/v{version}/",
"subjects":"http://na1.salesforce.com/services/data/v{version}/subjects/",
"search":"http://na1.salesforce.com/services/data/v{version}/search/",
"query":"http://na1.salesforce.com/services/data/v{version}/query/",
"profile":"http://na1.salesforce.com/005x0000001S2b9"},
"active":true,
"user_type":"STANDARD",
"language":"en_US",
"locale":"en_US",
"utcOffset":-28800000,
"last_modified_date":"2010-06-28T20:54:09.000+0000"}

```

After making an invalid request, the following are possible responses from Salesforce:

Error Code	Request Problem
403 (forbidden) — HTTPS_Required	HTTP
403 (forbidden) — Missing_OAuth_Token	Missing access token
403 (forbidden) — Bad_OAuth_Token	Invalid access token
403 (forbidden) — Wrong_Org	Users in a different organization
404 (not found) — Bad_Id	Invalid or bad user or organization ID
404 (not found) — Inactive	Deactivated user or inactive organization
404 (not found) — No_Access	User lacks proper access to organization or information
404 (not found) — No_Site_Endpoint	Request to the endpoint of a site
406 (not acceptable) — Invalid_Version	Invalid version
406 (not acceptable) — Invalid_Callback	Invalid callback

Setting a Custom Login Server

For special cases—for example, if you’re a Salesforce partner using Trialforce—you might need to redirect your customer login requests to a non-standard login URI. For iOS apps, you set the Custom Host in your app’s iOS settings bundle. If you’ve configured this setting, it will be used as the default connection.

Android Configuration

In Android, login hosts are known as server connections. Prior to Mobile SDK v. 1.4, server connections for Android apps were hard-coded in the SalesforceSDK project. In v. 1.4 and later, the host list is defined in the `res/xml/servers.xml` file. The SalesforceSDK library project uses this file to define production and sandbox servers.

You can add your servers to the runtime list by creating your own `res/xml/servers.xml` file in your application project. The root XML element for this file 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.)

Here’s an example of a `servers.xml` file.

```
<?xml version="1.0" encoding="utf-8"?>
<servers>
  <server name="XYZ.com Login" url="https://<username>.cloudforce.com"/>
</servers>
```

Server Whitelisting Errors

If you get a whitelist rejection error, you’ll need to add your custom login domain to the `ExternalHosts` list for your project. This list is defined in the `<project_name>/<platform_path>/config.xml` file. Add those domains (e.g. `cloudforce.com`) to the app’s whitelist in the following files:

For Mobile SDK 2.0:

- **iOS:** `/Supporting Files/config.xml`
- **Android:** `/res/xml/config.xml`

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.

Revoking Tokens

To revoke OAuth 2.0 tokens, use the revocation endpoint:

```
https://login.salesforce.com/services/oauth2/revoke
```

Construct a POST request that includes the following parameters using the `application/x-www-form-urlencoded` format in the HTTP request entity-body. For example:

```
POST /revoke HTTP/1.1
Host: https://login.salesforce.com/services/oauth2/revoke
Content-Type: application/x-www-form-urlencoded

token=currenttoken
```

If an access token is included, we invalidate it and revoke the token. If a refresh token is included, we revoke it as well as any associated access tokens.

The authorization server indicates successful processing of the request by returning an HTTP status code 200. For all error conditions, a status code 400 is used along with one of the following error responses.

- `unsupported_token_type`—token type not supported
- `invalid_token`—the token was invalid

For a sandbox, use `test.salesforce.com` instead of `login.salesforce.com`.

Handling Refresh Token Revocation in Android Native Apps

Beginning with Salesforce Mobile SDK version 1.5, native Android apps can control what happens when a refresh token is revoked by an administrator. The default behavior in this case 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, but they might or might not be suitable for your application. In your code you can choose whether to accept the default behavior or implement your own response. In either case, continue reading to determine whether you need to adapt your code.

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.

`TokenRevocationReceiver`, a utility class, is designed to respond to this intent action. To provide your own handler, you'll extend this class and override the `onReceive()` method. See [Token Revocation: Active Handling](#).

`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.

Token Revocation: Passive Handling

You can let the SDK handle all token revocation events with no active involvement on your part. However, even if you take this passive approach, you might still need to change your code. You do not need to change your code if:

- Your app contains any services, or
- All of your activities extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`.

If your app fails to satisfy at least one of these conditions, implement the following code changes.

1. (For legacy apps written before the Mobile SDK 1.5 release) In the `ClientManager` constructor, set the `revokedTokenShouldLogout` parameter to `true`.



Note: This step is not necessary for apps that are new in Mobile SDK 1.5 or later.

2. In any activity that does not extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`, amend the code as follows.
 - a. Declare a new variable:

```
private TokenRevocationReceiver tokenRevocationReceiver;
```

- b. In the `onCreate()` method add the following code:

```
tokenRevocationReceiver = new TokenRevocationReceiver(this);
```

- c. In the `onResume()` method add the following code:

```
registerReceiver(tokenRevocationReceiver, new  
IntentFilter(ClientManager.ACCESS_TOKEN_REVOKE_INTENT));
```

- d. In the `onPause()` method add the following code:

```
unregisterReceiver(tokenRevocationReceiver);
```

Token Revocation: Active Handling

If you choose to implement your own token revocation event handler, be sure to fully analyze the security implications of your customized flow, and then test it thoroughly. Be especially careful with how you dispose of cached user data. Because the user's access has been revoked, that user should no longer have access to sensitive data.

To provide custom handling of token revocation events:

1. The starting point for implementing your own response is the `SalesforceSDKManager.shouldLogoutWhenTokenRevoked()` method. By default, this method returns true. Override this method to return false in your `SalesforceSDKManager` subclass.

```
@Override
public boolean shouldLogoutWhenTokenRevoked() {
    return false;
}
```

2. The `ClientManager` constructor provides a boolean parameter, `revokedTokenShouldLogout`. Set this parameter to false. You can do this by calling `shouldLogoutWhenTokenRevoked()` on your `SalesforceSDKManager` subclass.
3. Implement your handler by extending `TokenRevocationReceiver` and overriding the `onReceive()` method.
4. Regardless of whether your activity subclasses `SalesforceActivity`, perform step 2 in [Token Revocation: Passive Handling](#).

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 administrators to set various security policies and have explicit control over who may use the corresponding applications.

A developer or administrator defines a connected app for Salesforce by providing the following information.

- Name, description, logo, and contact information
- A URL where Salesforce can locate the app for authorization or identification
- The authorization protocol: OAuth, SAML, or both
- Optional IP ranges where the connected app might be running
- Optional information about mobile policies the connected app can enforce

Salesforce Mobile SDK apps use connected apps to access Salesforce OAuth services and to call Salesforce REST APIs.

About PIN Security

Salesforce Connected Apps have an additional layer of security via PIN protection on the app. This PIN protection is for the mobile app itself, and isn't the same as the PIN protection on the device or the login security provided by the Salesforce organization.

In order to use PIN protection, the developer must select the **Implements Screen Locking & Pin Protection** checkbox when creating the Connected App. Mobile app administrators then have the options of enforcing PIN protection, customizing timeout duration, and setting PIN length.



Note: Because PIN security is implemented in the mobile device's operating system, only native and hybrid mobile apps can use PIN protection; HTML5 Web apps can't use PIN protection.

In practice, PIN protection can be used so that the mobile app locks up if it's isn't used for a specified number of minutes. When a mobile app is sent to the background, the clock continues to tick.

To illustrate how PIN protection works:

1. User turns on phone and enters PIN for the device.
2. User starts mobile app (Connected App).

3. User enters login information for Salesforce organization.
4. User enters PIN code for mobile app.
5. User works in the app, then sends it to the background by opening another app (or receiving a call, and so on).
6. The mobile app times out.
7. User re-opens the app, and the app PIN screen displays (for the mobile app, not the device).
8. User enters app PIN and can resume working.

Portal Authentication Using OAuth 2.0 and Force.com 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 Force.com 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 Force.com site.

Here's how to get started.

1. Associate a Force.com site with your portal. The site generates a unique URL for your portal. See [Associating a Portal with Force.com Sites](#).
2. Create a custom login page on the Force.com site. See [Managing Force.com 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.

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 Force.com site URL, such as `https://mysite.secure.force.com`:

```
https://mysite.secure.force.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 Force.com Developer Relations Blogs page.

Chapter 11

Distributing Mobile AppExchange Apps

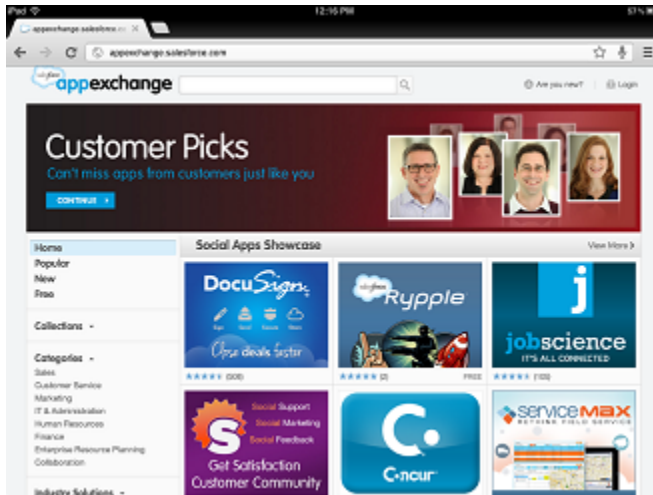
In this chapter ...

- [AppExchange for Mobile: Enterprise Mobile Apps](#)
- [Joining the AppExchange Partner Program](#)
- [Get a Publishing Org](#)
- [Create a Provider Profile](#)
- [The AppExchange Security Review](#)

Apps have completely redefined the mobile experience. When selecting a new smartphone or a tablet, consumers consistently rate app availability as the most important factor in their decision. So naturally, after you've developed your mobile app, you'll want to make it available so customers or staff can easily find, buy, and install it. Android and iOS have proprietary stores that list and distribute mobile apps, which won't be covered in this guide. Salesforce also has a marketplace called the AppExchange, where partners can list mobile apps and consulting services for Salesforce.

AppExchange for Mobile: Enterprise Mobile Apps

With almost half a million mobile app listings in consumer app stores, discovering the perfect enterprise app that is secure, trusted, and works within the Salesforce ecosystem can be a frustrating process. To help our customers find the perfect mobile app and to help developers reach millions of active Salesforce users, go to <http://www.appexchange.com>—the first cross-platform marketplace dedicated to enterprise mobile apps.



The AppExchange for Mobile connects developers with Salesforce users.

- Salesforce users can discover brand new mobile apps that are trusted, work with an existing account, and leverage data that's already in the cloud.
- ISVs can list their native, hybrid, and HTML5 applications that work on Android, iOS, and other platforms in a central repository. It doesn't matter whether the app is free, has a fixed price, or is sold with a subscription model.

Whether you're a developer that is working on a special purpose app that brings a unique mobile perspective for solving a specific problem, or a complete solution for a specific role, the space is completely open.

In order to distribute your commercial mobile app on AppExchange, you'll need to become a Salesforce partner.

1. Join the AppExchange Partner Program.
2. Log a case in the Partner Portal for a publishing org.
3. Create your Provider Profile on AppExchange.
4. Request a security review.
5. Log a case in the Partner Portal to request your app is listed on AppExchange for Mobile.



Note: If you're a Salesforce admin creating mobile apps for distribution within your organization, you don't need a public listing on the AppExchange.

Joining the AppExchange Partner Program

The first thing you need to do is join the AppExchange Partner Program. This program is designed to help independent software vendors (ISVs) be successful on the Salesforce platform.

1. In your browser go to www.salesforce.com/partners and click **Join Now**.
2. Select the first option: I want to build and market apps built on the Force.com platform (AppExchange Partner Program)

3. Answer questions about your application and your target market.
4. Fill in the fields about you and your company.
5. In the Additional Questions area, click the drop-down boxes and select the appropriate answer.
6. Enter the Captcha words shown and click **Submit Registration**.
7. In a moment you will receive an email with your username and temporary password. Click the link to the Partner Portal (<https://sites.secure.force.com/partners/PP2PartnerLoginPage>) and log in.
8. Accept the terms of use and then dismiss the pop-up that appears.
9. Bookmark this page, you'll be using it a lot.

In the Partner Portal you'll see quick links to some of the most used resources, and docs and video to get you started quickly. Most of this information is targeted at ISVs who create add-ons or services for Salesforce users. As a mobile ISV, you'll want to work closely with an AppExchange Partner Program representative.

Get a Publishing Org

In order to manage the distribution and support of your mobile app, you'll want to get a Salesforce organization that has full sales, marketing, and support functionality. This org is called the AppExchange Publishing Org, or APO for short. Qualified partners can get one for free through the Partner Portal.

1. In the Partner Community, under the Support tab, click **New Case**.
2. For the first category, choose **Orders and Contracts**.
3. For the second category, choose **Request ISV Business Org**.
4. In the Subject field, enter `Need ISV CRM`.
5. In the Description field, tell us if you have an existing org or if you need a new one. If you have an existing Salesforce org, you can provide the Org ID in the Description field and two additional CRM licenses will be added to your org. If you don't have an existing org, we'll provide a new one for you. In either case, make sure to enter your business address and then click **Submit Case**.
6. Shortly, you'll receive another email prompting you to log in and change your password. Do that, and then bookmark the page as before.

Create a Provider Profile

A provider profile represents your company on the AppExchange. You'll need to log into the Salesforce organization where you'll manage your business. If you're a qualified partner, you might already have an [APO org](#). If not, you can use the [Developer Edition](#) on page 14 org.

1. On the login page, use your username and password for your AppExchange Publishing Organization (APO).
2. Fill out the information in the Provider Profile and then click **Save**.

The AppExchange Security Review

Before you can list an app on AppExchange, you'll need to submit your app for a security review. The fastest way through the security review is to fully understand the security guidelines and process, which is online at http://wiki.developerforce.com/page/Security_Review.

The following procedure is for submitting packaged applications, but the steps are the same for mobile apps. After you submit the form, a representative will contact you for next steps.

1. Click **Start Review** on the Offering tab when editing the listing.

2. Select whether you charge for your application or if your application is free. Free applications must complete the review, but the review fee is waived.
3. If you charge for your application, Partner Operations will email you information within two business days on how to pay for the review. This is an annual payment.
4. Indicate if your application integrates with any web services outside of Force.com, including your own servers.
5. If your application integrates with other web services, list all of them in the **Webservices Used** box. You can enter up to 1000 characters.
6. If your application integrates with other web services, select how your application authenticates with those services. Enter any helpful comments in the box provided. You can enter up to 1000 characters.
7. Indicate if your application stores salesforce.com user credentials outside of Force.com.
8. Indicate if your application stores salesforce.com customer data outside of Force.com.
9. If your application stores salesforce.com customer data outside of Force.com, list all salesforce.com objects accessed in the **Objects Accessed** box. You can enter up to 255 characters.
10. Indicate if your application requires that customers install any client components, such as software or plug-ins.
11. If your application requires client components, enter the details in the **Requirements** box. You can enter up to 1000 characters.
12. Click **Start Security Review** to start the AppExchange Security Review. To discard your changes and return to the previous page, click **Cancel**.



Note: After collecting payment, the security team will send the partner a survey to collect detailed information on how to test their app. This will include information like install links, test credentials etc. that are mentioned in the next section.

13. You are contractually required to keep this information current. For example, if you upgrade your app to use a new web service, you must edit the information in your security review submission. To edit your submission information, click **Edit Review** on the Offering tab when editing the listing. Apps are reviewed again periodically.

Mobile apps have additional security steps, and you'll need to provide the following, depending on the phone type:

- iOS Mobile app — Provide the install link if the application is free and already published to the Appstore. If the application is not yet approved or is not free, please either provide an ad-hoc installation (contact us for device UDIDs), or a Testflight link for the app. (no UDID required). More information about Testflight is available at: <https://testflightapp.com/>. If credentials other than the Salesforce account login, or related external application credentials are required or optional for the mobile application, please provide them as well. If sample data is required for the application to function, please include a logical set of sample data.
- Android app — Provide the .APK for the android application and the target device. If credentials other than the Salesforce account login, or related external application credentials are required or optional for the mobile application, please provide them as well. If sample data is required for the application to function, please include a logical set of sample data.

Index

A

- About [4](#)
- Account Editor sample [240](#)
- AccountWatcher class [149](#)
- Android
 - FileRequests methods [153](#)
 - installing sample apps [24](#)
 - native classes [148](#)
 - push notifications [249](#)
 - push notifications, code modifications [250](#)
 - request queue [182](#)
 - RestClient class [152](#)
 - RestRequest class [152](#)
 - sample apps [25](#)
 - tutorial [164](#), [174–175](#)
 - WrappedRestRequest class [155](#)
- Android development [140](#), [144](#)
- Android hybrid development [83](#)
- Android project [141](#)
- Android requirements [141](#)
- Android sample app [179](#)
- Android template app [162](#)
- Android template app, deep dive [162](#)
- Android, native development [145](#)
- Apex controller [97](#)
- Apex REST resources, using [216](#)
- API access, granting to community users [258](#)
- API endpoints
 - custom [214](#)
- AppDelegate class [106](#)
- AppExchange [292–294](#)
- Application flow, iOS [105](#)
- application structure, Android [145](#)
- Audience [4](#)
- authentication
 - Force.com Sites
 - [291](#)
 - and portal authentication [291](#)
 - portal [291](#)
 - portal authentication [291](#)
- Authentication [278](#)
- Authentication flow [279](#)
- authentication providers [262](#)
- Authentication providers
 - Facebook [263–265](#)
 - Google [269](#)
 - Janrain [263–264](#)
 - OpenID Connect [269](#)
 - PayPal [269](#)
 - Salesforce [263–264](#), [267](#)
- Authorization [290](#)

B

- Backbone framework [200](#)
- Base64 encoding [151](#)
- BLOBs [197](#)
- Book version [5](#)
- Browsers
 - limited support [14](#)
 - recommendations [14](#)
 - requirements [14](#)
 - settings [14](#)
 - supported versions [14](#)

C

- caching data [186](#)
- caching, offline [204](#)
- Callback URL [18](#)
- Client-side detection [9](#)
- ClientManager class [151](#), [159](#)
- com.salesforce.androidsdk.rest package [159](#)
- Comments and suggestions [5](#)
- communities
 - add profiles [272](#)
 - API Enabled permission [271](#)
 - configuration [271](#)
 - configure for external authentication [275–276](#)
 - create a community [272](#)
 - create a login URL [272](#)
 - create new contact and user [272](#)
 - creating a Facebook app for external authentication [274](#)
 - Enable Chatter permission [271](#)
 - external authentication [262](#)
 - external authentication example [274–276](#)
 - external authentication provider [274](#)
 - Facebook app
 - [274](#)
 - example of creating for external authentication [274](#)
 - login endpoint [258](#)
 - Salesforce Auth. Provider [274–276](#)
 - testing [273](#)
 - tutorial [271–273](#)
- Communities
 - branding [259](#)
 - login [260](#)
- communities, configuring for Mobile SDK apps [256](#), [258](#)
- Communities, configuring for Mobile SDK apps [255–257](#)
- communities, granting API access to users [258](#)
- community request parameter [263](#)
- Comparison of mobile and PC [1](#)
- connected app
 - configuring for Android GCM push notifications [249](#)
 - configuring for Apple push notifications [251](#)
- connected app, creating [18](#)

- Connected apps 278, 290
- Consumer key 18
- Container 82
- create_native script 145
- Cross-device strategy 9
- custom endpoints, using 214

D

- data types
 - date representation 188
 - SmartStore 187–188
- Delete soups 189–190, 194–195
- designated initializer 131
- Detail page 93
- Developer Edition
 - vs. sandbox 12
- Developer.force.com 14
- Developing HTML apps 30
- Developing HTML5 apps 31, 80
- Development 13
- Development requirements, Android 141
- Development, Android 140, 144
- Development, hybrid 82, 95
- Distributing apps 292
- downloading files 181

E

- encoding, Base64 151
- Encryptor class 151
- endpoint, custom 214
- Enterprise identity 2–3
- Events
 - Refresh token revocation 288–289
- external authentication
 - using with communities 262
- external objects, using 219

F

- Feedback 5
- file requests, downloading 181
- file requests, managing 180–183, 185
- FileRequests methods 153
- Files
 - JavaScript 83
- files, uploading 181
- Flow 279–281
- Force.com 2
- Force.com for Touch 3
- Force.com, mobile services in 2
- Force.RemoteObject class 214
- Force.RemoteObjectCollection class 214
- ForcePlugin class 156

G

- Geolocation 2–3
- Getting Started 17
- GitHub 23
- Glossary 279

H

- HTML5
 - Getting Started 31
 - Mobile UI Elements 72
 - Mobile UI Elements sample app 75
 - using with JavaScript 31
- HTML5 development 7, 9, 16
- HTML5 development tools 31
- Hybrid Android development 83
- Hybrid applications
 - JavaScript files 83
 - Javascript library compatibility 95
 - Versioning 95
- Hybrid development 7, 9, 16, 82, 84, 90, 93, 95
- Hybrid guidelines 95
- Hybrid iOS sample 83
- Hybrid quick start 84
- Hybrid sample app 85

I

- Identity 3
- Identity services 2–3
- Identity URLs 282
- installation, Mobile SDK 20
- installing sample apps
 - Android 24
 - iOS 25–26
- Installing the SDK 21
- interface
 - KeyInterface 149
- Inventory 90, 93
- iOS
 - file requests 183
 - installing sample apps 25–26
 - push notifications 251
 - push notifications, code modifications 252
 - request queue 184
 - SFRestDelegate protocol 111
 - using SFRestRequest methods 114
 - view controllers 107
- iOS application, creating 102
- iOS apps
 - memory management 105
 - SFRestAPI 111
- iOS architecture 102, 141
- iOS development 101
- iOS Hybrid sample app 83

- iOS native app, developing [104](#)
- iOS native apps
 - AppDelegate class [106](#)
- iOS sample app [104](#), [139](#)
- iOS Xcode template [104](#)
- IP ranges [290](#)
- ISV [293–294](#)

J

- JavaScript
 - using with HTML5 [31](#)
- Javascript library compatibility [95](#)
- Javascript library version [97](#)
- JavaScript, files [83](#)

K

- KeyInterface interface [149](#)

L

- List page [90](#)
- localStorage [197](#)
- Location services [2–3](#)
- login and passcodes, iOS [105](#)
- LoginActivity class [155](#)

M

- MainActivity class [163](#)
- managing file download requests [181](#)
- managing file requests
 - iOS [183](#)
- Manifest, TemplateApp [164](#)
- memory management, iOS apps [105](#)
- Mobile Conatiner [2–3](#)
- Mobile container [82](#)
- Mobile Container [102](#)
- Mobile development [6](#)
- Mobile Development [102](#)
- Mobile inventory app [90](#), [93](#)
- Mobile policies [290](#)
- Mobile policy [2–3](#)
- Mobile SDK [2–3](#)
- Mobile SDK installation
 - node.js [20](#)
- Mobile SDK packages [20](#)
- Mobile SDK Repository [23](#)
- Mobile UI Elements
 - sample app [75](#)

N

- native Android classes [148](#)
- Native Android development [145](#)

- Native Android UI classes [155](#)
- Native Android utility classes [155](#)
- native API packages, Android [147](#)
- Native apps
 - Android [288](#)
- Native development [7](#), [9](#), [16](#)
- Native iOS application [102](#)
- Native iOS architecture [102](#), [141](#)
- Native iOS development [101](#)
- Native iOS project template [104](#)
- node.js
 - installing [20](#)
- npm [20](#)

O

- OAuth
 - custom login host [287](#)
- OAuth2 [278–279](#)
- offline caching [204](#), [207](#)
- offline management [186](#)
- Offline storage [187–188](#)
- Online documentation [4](#)

P

- Parameters, scope [281](#)
- Partner Program [293–294](#)
- PasscodeManager class [150](#)
- passcodes, using [156](#)
- PIN protection [290](#)
- Preface [1](#)
- Prerequisites [13](#)
- Printed date [5](#)
- project template, Android [162](#)
- Project, Android [141](#)
- push notifications
 - Android [249](#)
 - Android, code modifications [250](#)
 - iOS [251](#)
 - iOS, code modifications [252](#)
 - using [249](#)

Q

- Queries, Smart SQL [193](#)
- Querying a soup [189–190](#), [194–195](#)
- querySpec [189–190](#), [194–195](#)
- Quick start, hybrid [84](#)

R

- refresh token [98](#)
- Refresh token
 - Revocation [288–289](#)
- Refresh token flow [281](#)

- Refresh token revocation [288](#)
- Refresh token revocation events [288–289](#)
- registerSoup [189–190, 194–195](#)
- RegistrationHandler class
 - extending for Auth. Provider [276](#)
- Releases [23](#)
- Remote access [279](#)
- Remote access application [18](#)
- RemoteObject class [214](#)
- RemoteObjectCollection class [214](#)
- Request parameters
 - community [263](#)
 - scope [264](#)
- request queue, managing [182](#)
- request queue, managing, iOS [184](#)
- resource handling, Android native apps [157](#)
- Responsive design [9](#)
- REST API
 - supported operations [109](#)
- REST APIs [109](#)
- REST APIs, using [159](#)
- REST request [113](#)
- REST requests
 - files [180–183, 185](#)
- REST requests, iOS [113](#)
- RestAPIExplorer [139](#)
- RestClient class [151–152, 159](#)
- RestRequest class [152, 159](#)
- RestResponse class [159](#)
- Restricting user access [290](#)
- Revoking tokens [288](#)
- RootViewController class [108](#)

S

- Salesforce Auth. Provider
 - Apex class [276](#)
- Salesforce Mobile SDK [3](#)
- Salesforce Platform Mobile Services [2](#)
- Salesforce1 development
 - Salesforce1 vs. custom apps [4](#)
- Salesforce1 Platform [3](#)
- SalesforceActivity class [151](#)
- SalesforceSDKManager class [148](#)
- SalesforceSDKManager.shouldLogoutWhenTokenRevoked() method [288](#)
- SAML
 - authentication providers [263–265, 267, 269](#)
- Sample app, Android [179](#)
- Sample app, iOS [139](#)
- sample apps
 - Android [24–25](#)
 - iOS [25–26](#)
 - SmartSync [232](#)
- Sample hybrid app [85](#)
- Sample iOS app [104](#)

- sandbox org [12](#)
- Scope parameters [281](#)
- scope request parameter [264](#)
- SDK prerequisites [13](#)
- SDK version [97](#)
- SDKLibController [97](#)
- Security [278](#)
- Security review [294](#)
- Send feedback [5](#)
- Server-side detection [9](#)
- session management [98](#)
- SFRestAPI (Blocks) category, iOS [115](#)
- SFRestAPI (Files) category, iOS [118](#)
- SFRestAPI (QueryBuilder) category [116](#)
- SFRestAPI interface, iOS [111](#)
- SFRestDelegate protocol, iOS [111](#)
- SFRestRequest class, iOS
 - iOS [114](#)
 - SFRestRequest class [114](#)
- SFRestRequest methods, using [114](#)
- shouldLogoutWhenTokenRevoked() method [288](#)
- Sign up [14](#)
- Single sign-on
 - authentication providers [262](#)
- Smart SQL [187, 193](#)
- SmartStore
 - about [187](#)
 - adding to existing Android apps [188](#)
 - data types [187](#)
 - date representation [188](#)
 - enabling in hybrid apps [188](#)
 - soups [187](#)
- SmartStore extensions [197](#)
- SmartStore functions [189–190, 194–195](#)
- SmartSync
 - conflict detection [210, 212](#)
 - JavaScript [203](#)
 - model collections [201–202](#)
 - model objects [201](#)
 - models [201](#)
 - offline caching [204](#)
 - offline caching, implementing [206](#)
 - tutorial [198, 200, 220–221, 223–228](#)
 - User and Group Search sample [235](#)
 - User Search sample [237](#)
 - using in JavaScript [203](#)
- SmartSync Data Framework [186](#)
- SmartSync sample apps [232](#)
- SmartSync samples
 - Account Editor [240](#)
- soups [187](#)
- Soups [189–190, 194–195](#)
- Source code [23](#)
- Store [292–293](#)
- StoreCache [187, 207](#)

storing files [197](#)
 supported operations, REST API [109](#)

T

Template app, Android [162](#)
 template project, Android [162](#)
 TemplateApp sample project [162](#)
 TemplateApp, manifest [164](#)
 Terminology [279](#)
 Tokens, revoking [288](#)
 tutorial
 Android [174–175](#)
 conflict detection [212](#)
 SmartSync [198](#), [200](#), [220–221](#), [223–228](#)
 SmartSync, setup [220](#)
 tutorials
 Android [164](#), [173](#), [177](#)
 iOS [131](#), [133](#)
 Tutorials [31](#), [37](#), [45](#), [49](#), [56](#), [58](#), [61](#), [64](#), [66–68](#), [70–71](#), [119–123](#),
 [125–127](#), [129](#), [139](#), [164](#), [166–169](#), [171](#), [173](#), [179](#)

U

UI classes (Android native) [151](#)
 UI classes, native Android [155](#)

Uninstalling Mobile SDK npm packages [22](#)
 UpgradeManager class [155](#)
 uploading files [181](#)
 upsertSoupEntries [189–190](#), [194–195](#)
 URLs, indentity [282](#)
 User-agent flow [280](#)
 Utility classes, native Android [155](#)

V

Versioning [95](#)
 Versions [5](#)
 view controllers, iOS [107](#)

W

Warehouse schema [90](#), [93](#)
 What's New [26–28](#)
 When to use Mobile SDK [4](#)
 When to use Salesforce1 [4](#)
 WrappedRestRequest class [155](#)

X

Xcode project template [104](#)