Mobile and Offline Developer Guide

Version 63.0, Spring '25

Last updated: February 7, 2025

© Copyright 2000–2025 Salesforce, Inc. All rights reserved. Salesforce is a registered trademark of Salesforce, Inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

CONTENTS

Chapter 1: Mobile App Development with Lightning Web Components and LWC Offline . . . 1

About This Developer Guide . . . 2
LWC Offline Enabled Apps . . . 2
Learn Lightning Web Components . . . 2
Learn Lightning Web Components with Trailhead . . . 3
Learn Lightning Web Components with Documentation . . . 4

Chapter 2: Create Mobile-Ready Components . . . 5

Understand Mobile Development . . . 6
Use Built-In Mobile Tools and Features . . . 6
Validate Your Base Component References . . . 7
Use Mobile Validation with Salesforce Lightning Design System . . . 9
Minimize Bandwidth Usage . . . 11
Create Responsive Layouts . . . 12
Follow Accessible Mobile Design Guidelines . . . 16
Disable Pull-to-Refresh in the Salesforce Mobile App . . . 19

Chapter 3: Use Mobile Device Features in Mobile Apps . . . 22

Request App Review . . . 24
AppReviewService User Experience . . . 24
Use the AppReviewService API . . . 27
AppReviewService Example . . . 28
Compatibility and Requirements . . . 30
Considerations and Limitations . . . 30
Scan Barcodes . . . 31
Barcode Scanning User Experience . . . 32
Use the BarcodeScanner API . . . 34
BarcodeScanner Example–Modern Scanning API . . . 38
BarcodeScanner Example—Single Scan (Legacy) . . . 40
Scan Multiple Barcodes (Legacy) . . . 43
BarcodeScanner Example—Continuous Scanning (Legacy) . . . 45
Create a Self-Service Kiosk Application . . . 48
BarcodeScanner Example—Self-Service Kiosk (Legacy) . . . 51
Customize the BarcodeScanner User Interface . . . 54
Compatibility and Requirements . . . 56
Considerations and Limitations . . . 57
Access Device Biometrics . . . 58
BiometricsService User Experience . . . 59

Use the BiometricsService API . . . 59
BiometricsService Example . . . 61
Compatibility and Requirements . . . 62
Considerations and Limitations . . . 62
Manage Calendar Events . . . 63
CalendarService User Experience . . . 63
Use the CalendarService API . . . 64
CalendarService Example . . . 67
Compatibility and Requirements . . . 77
Considerations and Limitations . . . 77
Access Contacts . . . 78
ContactsService User Experience . . . 79
Use the ContactsService API . . . 80
ContactsService Example . . . 82
Compatibility and Requirements . . . 84
Considerations and Limitations . . . 84
Scan Documents on a Mobile Device . . . 86
Use the DocumentScanner API . . . 86
DocumentScanner Example . . . 88
Compatibility and Requirements . . . 91
Considerations and Limitations . . . 92
Monitor Geofence Regions on a Mobile Device . . . 93
GeofencingService User Experience . . . 94
Use the GeofencingService API . . . 94
GeofencingService Example . . . 97
Compatibility and Requirements . . . 98
Considerations and Limitations . . . 98
Use Location . . . 99
LocationService User Experience . . . 100
Location Basics . . . 101
Use the LocationService API . . . 101
LocationService Example . . . 103
Compatibility and Requirements . . . 106
Considerations and Limitations . . . 107
Interact with NFC Tags on a Mobile Device . . . 108
NFCService User Experience . . . 108
Use the NFCService API . . . 109
NFCService Example . . . 110
Compatibility and Requirements . . . 113
Considerations and Limitations . . . 113
Accept On-Site Payments with Tap-to-Pay . . . 114
PaymentsService User Experience . . . 115
Use the PaymentsService API . . . 115
PaymentsService Example . . . 117

Compatibility and Requirements . . . 120
Considerations and Limitations . . . 120

Chapter 4: Offline Considerations and Limitations . . . 121

General Considerations . . . 122
Considerations for Field Service Mobile App . . . 122
Base Components Support . . . 123
Modules Support . . . 125
Wire Adapters Support . . . 128
Entity Support . . . 131
Metadata and Custom Metadata Types Support . . . 132

Chapter 5: Offline Environment Details . . . 133

What Happens When Something Isn't Primed (Preloaded) . . . 134
Create Components with Offline Analysis In Mind . . . 134
Determine Online or Offline Status . . . 135

Chapter 6: Use Salesforce Features While Offline . . . 136

Use GraphQL While Mobile and Offline . . . 137
Understand Salesforce GraphQL Implementations . . . 137
Feature Limitations of Offline GraphQL . . . 138
Best Practices for Using GraphQL in LWC Offline . . . 140
Use Apex While Mobile and Offline . . . 141
Use Apex in Lightning Web Components While Online . . . 142
Enable Caching of Apex Results . . . 143
Apex Results Are Separate from Other Primed Data . . . 144
Understand Apex Behavior While Offline . . . 145
Additional Considerations for Apex in an Offline-Enabled Mobile App . . . 147
Additional Documentation for Apex in Lightning Web Components . . . 147
Use Images in an LWC Offline-Enabled Component . . . 147
Use Images Uploaded as Files (ContentDocument) in an LWC . . . 148
Use Images Uploaded as Asset Files . . . 150
Use Images Uploaded as Static Resources . . . 151
Image Priming and Offline Considerations . . . 152
Upload Images While Offline . . . 153
Understand File Uploads in Salesforce . . . 153
Image Upload Basics . . . 154
Image Upload Example . . . 154
Use Third-Party JavaScript in an LWC Offline-Enabled Component . . . 160
Navigation . . . 161
Navigation User Experience . . . 162
Base Components with Built-In Navigation Actions . . . 162
Programmatic Navigation Actions . . . 163

Chapter 7: Development Tools and Processes . . . 170

Understand the Mobile Development Cycle . . . 171
Set Up Your Development Environment . . . 172
Set Up Xcode . . . 174
Set Up Android Studio . . . 175
Install Mobile Extensions . . . 180
Preview Components on Mobile . . . 184
Mobile Development Preview Environments . . . 184
Preview from the Command Line . . . 188
Preview from VS Code . . . 191
Preview in the Salesforce Mobile App . . . 192
Preview in Custom Mobile Apps . . . 194
Validate Lightning Web Components for Offline Use . . . 196
Install the Komaci Static Analyzer . . . 196
Troubleshoot Installation Problems . . . 197
Validate Components During Development . . . 198
Static Analyzer Validation Rules . . . 198
Install ESLint Rules for Mobile Lightning Web Components . . . 199
Develop Offline-Ready LWCs with the LWC Offline Test Harness . . . 200
Test Harness Overview . . . 201
Install the Test Harness App . . . 210
Use the Test Harness App . . . 212
Debug Lightning Web Components . . . 213
Debug Mobile Components . . . 216
Customize the Offline Experience for the Salesforce Mobile App . . . 217
Prerequisites & Setup Considerations . . . 218
Download and Install . . . 218
Configure the Offline Experience . . . 219

Chapter 8: Quick Start Tutorials . . . 225

Develop a Lightning Web Component Quick Action . . . 226
Prerequisites . . . 226
Field Service Org Setup . . . 226
iOS Simulator Setup . . . 228
Android Emulator Setup . . . 229
Workspace Setup . . . 232
Create and Configure a Lightning Web Component . . . 233
Debug Lightning Web Components in the Field Service Mobile App . . . 239
Install Local Development Server Plugin . . . 239
Debug in iOS . . . 240
Debug in Android . . . 241

CHAPTER 1 Mobile App Development with Lightning Web Components and LWC Offline

In this chapter ...

- About This Developer Guide
- LWC Offline Enabled Apps
- Learn Lightning Web Components

Customize Salesforce mobile apps with features built with Lightning web components, and deploy your customizations to mobile users. Create components and apps that work even when mobile devices are offline while in the field. Optimize your features to handle low- and no-network connectivity situations with grace.

LWC Offline is an advanced runtime environment for Lightning web components. Available only for mobile devices, it replaces the standard Lightning components runtime, and augments it with features designed specifically for mobile and offline use.

- Briefcase Builder lets you define advanced data priming strategies, customized for the objects and records that your users need access to while offline.
- A new priming engine preloads records when you prepare to go offline.
- A durable on-device cache holds primed records for offline access, including changes made while offline.
- Lightning Data Service (LDS) is enhanced to work seamlessly with primed records. While online, LDS uses the cache as a performance enhancer. While offline, LDS allows transparent access to existing, changed, and even new records.

There are many, many other changes that you (mostly) don’t need to worry about. The overall goal for LWC Offline is to let you develop Lightning web components that “just work” whether you’re online or offline.

About This Developer Guide

Documentation for LWC Offline is a work in progress, and will improve continuously throughout the pilot and beta programs. The initial release of the program documentation is shared across several LWC Offline-enabled mobile apps. Use this developer guide for the purpose of evaluating LWC Offline in your own orgs.

Where possible, documentation refers to LWC Offline features generally, and applies to all LWC Offline-enabled mobile apps. However, these efforts are in progress and incomplete. You'll likely notice references to Field Service in the documentation, and some instructions might be specific to Field Service. In the early stages of this pilot, we'll ask for your forbearance and forgiveness.

In the meantime, if you're unable to interpret these differences, don't hesitate to reach out with questions on the relevant Trailblazer community.

LWC Offline Enabled Apps

LWC Offline is available as an optional, opt-in enhancement to existing Salesforce mobile apps.

The following apps can use LWC Offline when the feature is enabled for your org.

- **Salesforce mobile app**

  Available as a GA feature.

  Note: Your organization must purchase and license Salesforce Mobile App Plus in order to use Mobile Offline. Contact your Salesforce sales rep for more information.

- **Field Service mobile app**

  Available as GA feature that any Field Service org can opt into. Join the Trailblazer community for access to additional resources.

SEE ALSO:

Salesforce Help: Salesforce Mobile App Plus

GitHub: Offline App Developer Starter Kit

Learn Lightning Web Components

To create Lightning web components for use in LWC Offline-enabled mobile apps, you must learn the basics of Lightning web components.

Lightning Web Components (the name of the framework) was introduced in 2018, and represents the best, most performant, and most modern framework for building custom apps for Salesforce. Learning Lightning web components lets you build apps for desktop, mobile online, and mobile offline environments.

IN THIS SECTION:

Learn Lightning Web Components with Trailhead

If you're not already an experienced LWC developer, the best way to learn Lightning web components is with the extensive collection of lessons and projects on Trailhead.

Learn Lightning Web Components with Documentation

Use the *Lightning Web Component Developer Guide* to understand the Lightning Web Components framework and how to use it with Salesforce.

Learn Lightning Web Components with Trailhead

If you're not already an experienced LWC developer, the best way to learn Lightning web components is with the extensive collection of lessons and projects on Trailhead.

Comprehensive Trailhead Trails

For the most complete foundation, start with these Trailhead trails to get you up to speed with Lightning web components, including necessary JavaScript skills.

- Learn to Work with JavaScript

  Lightning web components are implemented using modern HTML, JavaScript, and CSS. We suggest that you have **at least** an intermediate level of skill with JavaScript. This trail consists of two modules that make sure your JavaScript background is solid.

- Build Lightning Web Components

  This Trailhead trail provides a complete foundation for working with Lightning web components.

Essential Trailhead Modules

To focus only on the essentials, these Trailhead modules provide the conceptual basics of Lightning web component development. Every developer working with LWCs should complete **all** of these modules to ensure they have solid, basic skills.

- Lightning Web Components Basics
- Lightning Web Components and Salesforce Data
- Communicate Between Lightning Web Components

Focused Trailhead Projects

These projects get you going *fast* with Lightning web components. Fast is exciting!

Note: Most LWC modules and projects are general purpose, rather than specific to a particular mobile app. For setting up to specifically work with Lightning web components and your specific mobile app, such as the Field Service mobile app, follow the steps provided in Set Up Your Development Environment on page 172.

- Quick Start: Lightning Web Components
- Set Up Your Salesforce Mobile Developer Tools for Lightning Web Components
- Set Up Your Lightning Web Components Developer Tools

MOAR, MOAR, MOAR

These modules and projects are great if you want to learn specific areas of Lightning web components, or have experience with our other UI customization frameworks, such as Visualforce or Lightning Aura components.

- Lightning Web Component Troubleshooting
- Lightning Web Components Tests
- Lightning Web Components for Aura Developers
- Lightning Web Components for Visualforce Developers
- Build a Bear-Tracking App with Lightning Web Components

Learn Lightning Web Components with Documentation

Use the *Lightning Web Component Developer Guide* to understand the Lightning Web Components framework and how to use it with Salesforce.

The *Lightning Web Component Developer Guide* is the canonical resource for all details of developing with Lightning web components. After you've learned the conceptual basics, you continue to use the developer guide to find answers to specific "how do I...?"-type questions. The developer guide also includes the Component Reference, which provides reference documentation for all of the built-in base components available in the framework.

Mobile-specific documentation for Lightning web components includes the following:

- Preview Lightning Web Components on Mobile (full details on development tools)
- Create Mobile-Ready Components (guidelines and best practices)
- Use Mobile Device Features in Mobile Apps (add platform native features to LWCs)

While not specific to Lightning web components or LWC Offline, the following documentation can be helpful during your development efforts.

- Field Service Developer Guide
- Field Service Mobile App Salesforce Help
- Briefcase Builder Salesforce Help
- Quick Actions Salesforce Help
- Salesforce Mobile Debugging Tools
- Field Service Mobile LWCs Trailblazer community
- Mobile Automated Testing Trailblazer community

CHAPTER 2 Create Mobile-Ready Components

In this chapter ...

- Understand Mobile Development for Lightning Web Components
- Use Built-In Mobile Tools and Features
- Minimize Bandwidth Usage
- Create Responsive Layouts
- Follow Accessible Mobile Design Guidelines
- Disable Pull-to-Refresh in the Salesforce Mobile App

Build components that perform well across mobile experiences. These guidelines are best practices, not universal rules. Consider them carefully, but don't be afraid to go your own way if there are compelling reasons in specific situations.

Understand Mobile Development for Lightning Web Components

The best way to start building mobile-ready components is to deep dive into Lightning Web Components first.

Accelerate your mobile development journey with these resources.

- Build Lightning Web Components

  Start with learning how to build Lightning web components before you can build mobile-ready Lightning web components.

- Set Up Your Salesforce Mobile Developer Tools for Lightning Web Components

  Learn how to configure your local workspace with the tools needed for developing and testing your mobile-ready Lightning web components.

- Quick Start: Salesforce DX and App Development with Salesforce DX

  Learn about modern development tools for developing on the Salesforce platform. These tools are essential for being successful with Lightning web components.

- Transform Your Business with Mobile

  Learn how mobile apps can transform your business and improve employee productivity.

- Develop with Mobile SDK

  Create your own iOS, Android, or hybrid mobile apps powered by the Salesforce Platform.

- Developer Intermediate

  Take your apps to the next level with powerful integration and mobile tools.

- **Video:** Salesforce Mobile Tools

  Install and configure your mobile tools with follow-along videos.

Use Built-In Mobile Tools and Features

Before you write code for your users on mobile devices, configure your environment and use built-in mobile-ready tools and features.

To develop Lightning web components that are optimized for mobile, follow these prerequisites and processes.

- Set up your development environment

  We recommend setting up the VS Code editor and the Salesforce DX tools for an end-to-end development experience. You'll also want to set up a local development server and other tools that can help with mobile development.

- Install the mobile extensions plug-in for VS Code

  Inspect and preview your components on virtual mobile devices before you deploy them to mobile users.

- Install the SLDS validator for VS Code

  The SLDS validator enables syntax highlighting and code completion with intelligent token and utility class recommendations. The validator optimizes your component styling and helps you build components across screen sizes without having to memorize all the SLDS guidelines. It also checks for usage of base components that aren't deemed mobile ready and suggests replacement options.

  - Validate Your Base Component References
  - Use Mobile Validation with Salesforce Lightning Design System

After you create a Lightning web component:

- Validate the component in mobile preview environments

  The preceding list of prerequisites prepares your environment for mobile previews, which helps you validate your visual changes and other basic mobile behavior.

- Configure your component for Lightning App Builder

  Configuring your component for Lightning App Builder allows an admin or business user to use the component when they create or customize Lightning app pages and record pages. Alternatively, you can surface your Lightning web component via a custom tab instead.

- Create a Lightning app page and add the component to your mobile navigation

  Enable your components for Lightning App Builder to allow admins to create Lightning pages with your components. Lightning app pages and record pages are supported for mobile experiences, such as the Salesforce mobile app and custom mobile apps. If you surface your component in a custom tab instead, make it available to the mobile app via App Manager or Salesforce Navigation in Setup.

IN THIS SECTION:

Validate Your Base Component References

Base components help you develop apps quickly. However, not all base components are designed for mobile environments. The SLDS Validator for VS Code can help you determine the mobile readiness of the base components you use.

Use Mobile Validation with Salesforce Lightning Design System

The Salesforce Lightning Design System (SLDS) validator checks your code for SLDS mobile guidelines adherence as you type. If it finds a potential issue, the validator provides a warning with suggested improvements. These warnings apply to HTML and CSS code.

Validate Your Base Component References

Base components help you develop apps quickly. However, not all base components are designed for mobile environments. The SLDS Validator for VS Code can help you determine the mobile readiness of the base components you use.

What Are Mobile-Ready Components?

A component is considered mobile-ready if it meets the following conditions:

- The component renders correctly when displayed on a mobile device:
  - The component responds to fit within the reduced screen size.
  - The layout of component elements and controls remains sensible.
- The component doesn't require interactions that are awkward when performed using touch-based input. For example, side-to-side scrolling is awkward or not supported on a narrow phone screen.
- The component doesn't require constant connections to a server-side controller as you interact with it. For example, auto-suggest look-ups can require a new server request with every search term change you type.
- There are no known issues when the component is used in a non-desktop browser.

The SLDS validator uses the same list of mobile-ready components used by the Component Reference. You can find documentation, examples, and specifications for all base components in the Component Reference of the Lightning Component Library. Select a component, then click **Example**, **Documentation**, or **Specifications**.

Install the SLDS Validator

Base component validation requires version 1.4.4 or later of the SLDS Validator for Visual Studio Code.

Enable or Disable the SLDS Validator

You can globally enable or disable the SLDS validator. In VS Code Preferences, go to **Settings** > **Extensions** > **SLDS Validator** > **Salesforce-vscode-slds** > **Basic: Mobile Validation**.

Validate Base Components in Your Code

While you're coding, the SLDS validator places markers on components that are known to have mobile issues. For example, the validator checks all base component references against its "mobile ready" list. This list shows components that function as well on a limited mobile screen as on a desktop. If the component isn't on the list, the validator highlights the code that uses that component with a yellow underline. Any base component reference that doesn't show the yellow underline is ready to go mobile.

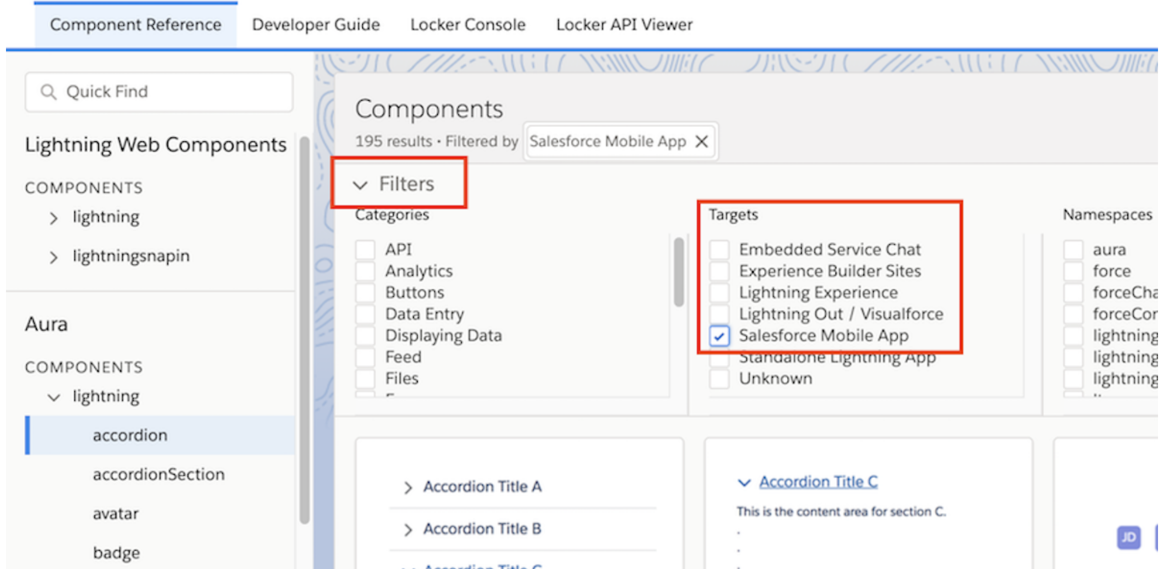
To read details about the warning, hover your mouse over the highlighted code.

```
1   <template>
2       <lightning-card title="HelloWorld" icon-name="custom:custom14">
3           <div class="slds-card__body slds-card__body_inner">
4               Hello, {name}!
5           </div>
6       </lightning-card>
7       <lightning-datatable></lightning-datatable>
8   </te lightning-datatable is known to have issues on mobile devices.
        Consider these options:
        • Find a mobile-ready replacement in the Component Reference of
        the Lightning Web Components Dev Guide. Filter the list using
        "Salesforce Mobile App" as the target.
        • Create a custom component that complies with "Follow Accessible
        Mobile Design Guidelines" in the Lightning Web Components Dev
        Guide. SLDS(4)
        A table that displays rows and columns of data. View in Component Library
```

Resolve Warnings

To avoid or address validation warnings, consider these options.

- Use only mobile-ready base components. To find these components:
  - Open the Component Reference.
  - Expand Filters.
  - Under Targets, choose **Salesforce Mobile App**. The filtered list shows only mobile-ready components.



- Update your existing Lightning web components to be mobile-ready. See:
  - Follow Accessible Mobile Design Guidelines
  - Lightning Design System: Accessible Mobile Design Guidelines
- If you're creating your first mobile-ready component, start with Create Mobile-Ready Components.

Use Mobile Validation with Salesforce Lightning Design System

The Salesforce Lightning Design System (SLDS) validator checks your code for SLDS mobile guidelines adherence as you type. If it finds a potential issue, the validator provides a warning with suggested improvements. These warnings apply to HTML and CSS code.

SLDS defines a wide range of tokens that standardize user interface style descriptors and units. For example, you can use SLDS tokens to specify text style attributes such as font, font size, and font color. Tokens make it easy to ensure that your design is consistent, and simplify updates as your design evolves.

SLDS provides its own validator extension for Visual Studio Code. The SLDS validator scans your Lightning web component code looking for expressions that stray from SLDS guidelines. If it finds issues, the validator suggests an appropriate SLDS token or provides other advice for improving the underlined expression.

For mobile readiness, the validator performs additional checks that address mobile accessibility. Accessibility on mobile devices demands stricter guidelines than on desktop browsers, for the benefit of all users. On smaller phone screens, for example, fonts that fall below certain size thresholds can be difficult to read even for customers with excellent eyesight. Word wrapping also becomes more important on limited screen sizes that don't support horizontal scrolling. SLDS validator warnings keep you informed when your settings appear to degrade a component's mobile effectiveness.

SLDS Guidelines for Mobile Accessibility

Warning messages for mobile readiness are based on the following SLDS guidelines.

Font size

To improve mobile readability, use an SLDS token from the following value range. If you must use another unit type, choose a value from one of the equivalent px, pt, em, rem, or % ranges:

Unit	Recommended value range
SLDS token (recommended)	`$font-size-4` or larger. See "Font Size" in Lightning Design System.
px	14 px or larger
pt	10.5 pt or larger
em, rem	0.875 or larger
%	87.5 or larger

For specific use case recommendations, see Follow Accessible Mobile Design Guidelines on page 16.

Word wrapping in labels

To avoid truncation at runtime, always use word wrapping in labels. Avoid using the ellipsis.

Clickable images, Button icons, form elements

To clarify UI behavior, always provide labels for visual elements that support user interaction.

Install the SLDS Validator

SLDS mobile validation requires version 1.4.7 or later of the SLDS Validator for Visual Studio Code.

Enable or Disable the SLDS Validator

You can globally enable or disable the SLDS validator. In VS Code Preferences, go to **Settings** > **Extensions** > **SLDS Validator** > **Salesforce-vscode-slds** > **Basic: Mobile Validation**.

Use SLDS Warnings to Validate Your Code

While you're coding, the SLDS validator places markers on components that don't conform to mobile guidelines. For example, if an element of your component uses a text size smaller than `$font-size-4`, the validator highlights the unsuitable font size with yellow underlining.

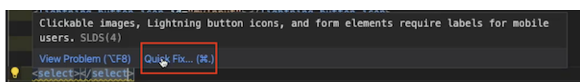
To read details about the warning, hover your mouse over the highlighted code.

Suppress Selected Mobile Readiness Warnings

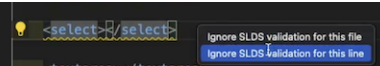
Sometimes, you have a good reason to ignore a warning—for example, you're at a stage of development where you're focusing only on the "big picture". In such cases, you can suppress SLDS warnings using the Quick Fix menu.

In HTML or CSS files:

1. Hover over an element that's underlined in yellow. In the warning window, click Quick Fix.



2. From the Quick Fix submenu, choose to suppress either the warning for the current line only, or all SLDS warnings for the current file.



To remind itself—and you—of suppressed mobile readiness warnings, the SLDS validator inserts code comments. If you'd like to pre-empt these warnings manually in HTML files, you can insert the comments yourself as follows:

- Any content that is between the following pair of lines is exempt from SLDS validation:

```
<!-- sldsValidatorIgnore -->
...
<!-- sldsValidatorAllow -->
```

- The following line exempts the immediate next line from SLDS validation:

```
<!-- sldsValidatorIgnoreNextLine -->
```

- If the line `<!-- sldsValidatorAllow -->` doesn't exist elsewhere in the HTML file, the following line at the top of the file exempts the file's entire content from SLDS validation:

```
<!-- sldsValidatorIgnore -->
```

SEE ALSO:

SLDS: Accessible Mobile Design Guidelines

SLDS: Design Tokens

W3C CSS Tips and Tricks

Minimize Bandwidth Usage

Since mobile users have network constraints, consider bandwidth on mobile devices when building your components.

Minimize CSS and JavaScript Libraries

Having many CSS and JavaScript resources can impact loading time. To improve load time, minimize your CSS and JavaScript libraries. Remove comments and whitespace, and compress the resources before you upload your files as static resources. Images and other assets served this way benefit from the caching and content distribution network (CDN) built into Salesforce. See Access Static Resources.

A Lightning web component's JavaScript file can have a maximum file size of 128 KB (131,072 bytes). To work with third-party JavaScript libraries, consider building custom versions of JavaScript libraries with only the functions you need. Many open-source JavaScript libraries provide this option, which substantially reduces the file size. See Use Third-Party JavaScript Libraries.

Reduce Image Size

Loading large or high-resolution image files can significantly affect performance. Use fewer images and smaller background textures, and use CSS instead of images when possible. If you must work with multiple or large images, reduce image download size by compressing image files 10–30% using image compression tools.

Consider using CSS sprites instead of individual images. CSS sprites combine a group of similarly sized graphics, such as buttons and icons, into a single file. To display parts of the combined image, use the CSS `background-image` and `background-position` properties. Reducing the number of image files used minimizes the number of HTTP requests. Also, the combined sprite file of images is easily cached, and therefore more performant for all devices.

Prioritize vector graphics, also known as SVG images, as they are often smaller in file size, and scale efficiently at any screen sizes. To use a raster image, such as a JPEG or PNG file, follow general responsive design guidelines and consider provisioning raster images only for high-resolution screens. Check the screen resolution for the devices you plan to support. Prepare an image for various resolutions and serve the best quality image corresponding to the screen size. See Use SVG Resources.

Follow General Development and Offline Management Best Practices

To improve overall bandwidth usage, follow development best practices. Present a simple landing page to your mobile users and link to more complex components or pages later. Similarly, lazy load the page to allow basic HTML to load first before loading custom libraries.

Since mobile connectivity issues are common, we recommend that your app handles offline scenarios gracefully. Offline access is available in Salesforce for Android and iOS. When you cache data to make it available offline, mobile users get better overall performance and faster access to previously accessed records. See Enable Offline Access and Offline Edit for the Salesforce Mobile App. If you build a custom mobile app using the Mobile SDK, consider storing and synchronizing data for offline use. See Offline Management.

We recommend following the Lightning Design System design guidelines for a consistent experience. When you expect a noticeable lag when loading a page, use loading indicators, such as a spinner (`lightning-spinner`) or another animated SVG or GIF image. Use lightweight stencils when data takes longer than 300 ms to retrieve. See Loading in the SLDS Design Guidelines.

SEE ALSO:

MDN: Responsive Images

Create Responsive Layouts

For a responsive, mobile-first app, create layouts using the grid system.

Responsive design enables your app page or website to scale elegantly across screen sizes. It uses *fluid grids* and *media queries* to display the right layout for different screens. Responsive design improves app maintainability and reliability, while future proofing for different browsers and platforms.

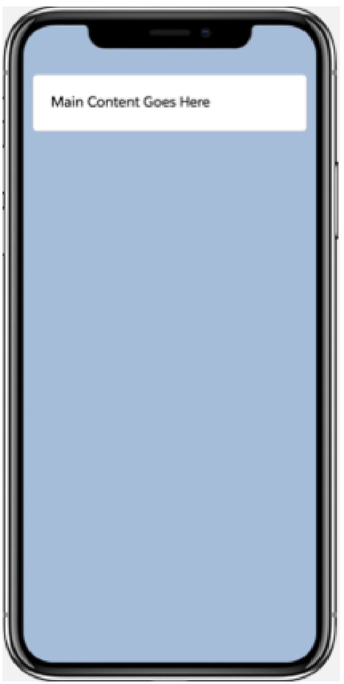
Note: To vary functionality across devices, or to create a mobile version of a large existing site, consider creating a separate mobile site or app. In these cases, creating something new may be easier that creating a responsive layout.

The first step in creating a responsive layout is to implement a fluid grid. The SLDS grid utility provides a mobile-first layout system with granular column control. To implement the grid system in a Lightning web component, use the `lightning-layout` and `lightning-layout-item` base components. When implementing fluid grids, the layout starts to break down at specific breakpoints. To resolve this layout issue, media queries determine how the layout should look at each breakpoint. These responsive breakpoints are built into `lightning-layout-item`, enabling you to define how each column adjusts to the screen size.

Let's start with a one-column grid. Despite being simple visually, in this example there are three separate levels of containment. The `lightning-card` base component defines a container with rounded corners around the content. The `lightning-layout` base component creates a wrapper around the content using the `slds-grid` class. The `lightning-layout-item` component creates columns using the `slds-col` class.

```
<!-- Page header here -->
<!-- Start page content -->
<lightning-card>
    <lightning-layout>
        <lightning-layout-item padding="around-small">
            <p>Main Content Goes Here</p>
        </lightning-layout-item>
   </lightning-layout>
```

```
</lightning-card>
<!-- End page content -->
```



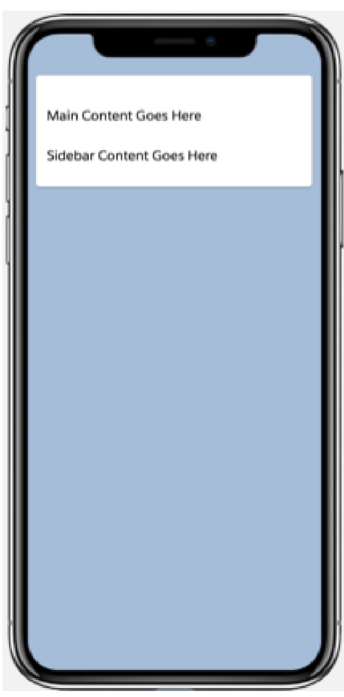
To increase the number of columns, add another `lightning-layout-item` with some content.

```
<lightning-card>
   <lightning-layout>
      <lightning-layout-item padding="around-small">
        <p>Main Content Goes Here</p>
      </lightning-layout-item>
      <lightning-layout-item padding="around-small">
          <p>Sidebar Content Goes Here</p>
      </lightning-layout-item>
  </lightning-layout>
</lightning-card>
```

Let's make some adjustments so that the content is more readable on mobile. To make each column full width, add `size="12"`. The 12-column grid is the most frequently used SLDS grid. On this grid, the `size` attribute for `lightning-layout-item` accepts a value from 1 to 12.

The multiple-rows attribute on `lightning-layout` adds a `slds-wrap` class to the container, which wraps your column to a new row.

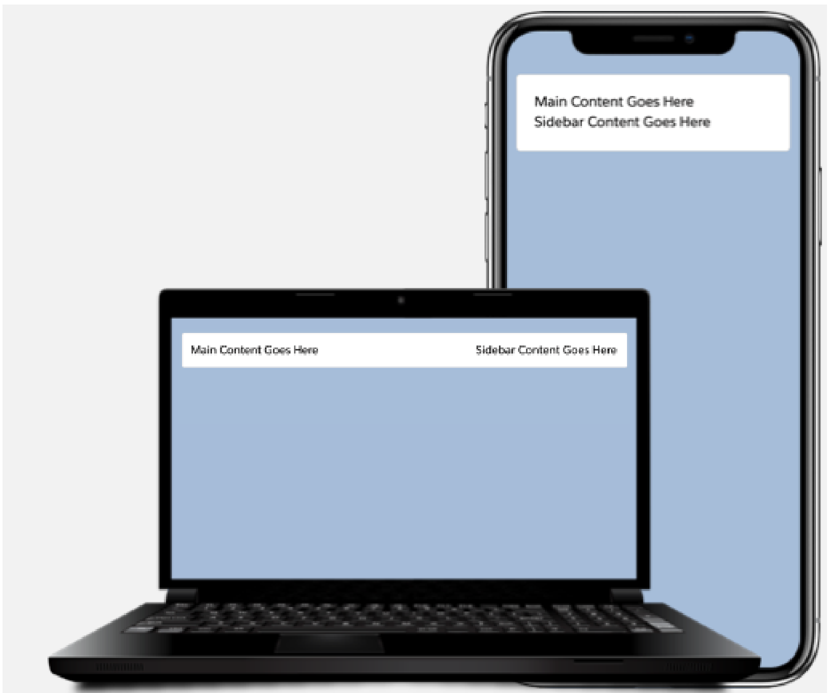
```
<lightning-card>
   <lightning-layout multiple-rows>
    <lightning-layout-item size="12" padding="around-small">
      <p>Main Content Goes here</p>
     </lightning-layout-item>
     <lightning-layout-item size="12" padding="around-small">
       <p>Sidebar Content Goes Here</p>
    </lightning-layout-item>
  </lightning-layout>
</lightning-card>
```



When we view the page on a tablet or desktop, the sidebar column should align horizontally with the main content column. Specify the width of the main content such that it takes up 75% of the container width using `small-device-size="9"`. See `lightning-layout-item`.

Using *just markup*, you can provide size details for both desktop/tablet and phone/small mobile. It's ratio-based and enables the device to effectively pick the right layout based on its own screen size.

```
<lightning-card>
  <lightning-layout multiple-rows>
     <lightning-layout-item size="12"
          small-device-size="9"
          padding="around-small">
          <p>Main Content Goes Here</p>
      </lightning-layout-item>
      <lightning-layout-item size="12"
          small-device-size="3"
          padding="around-small">
          <p>Sidebar Content Goes Here</p>
      </lightning-layout-item>
  </lightning-layout>
</lightning-card>
```



Use Your Component on a Lightning Page

We walked through creating a Lightning web component with a responsive layout. What if you want to make your component available in Lightning Experience or the Salesforce mobile app? You can surface your Lightning web component via a Lightning page or custom tab.

Lightning pages in your org support desktop and mobile form factors by default. Therefore, it's important that your components follow a responsive design so they work seamlessly across devices.

We recommend that you use the Lightning App Builder to build Lightning pages. In Lightning App Builder, you can select a template with the layout you want and drag your custom components onto the page. The template you choose controls how the page displays on a device. The structure of a Lightning page adapts for the device it's viewed on. See Lightning Pages in Salesforce Help.

If you don't find a template you want, create a custom template using an Aura component. See Create a Custom Lightning Page Template Component. Creating a custom template using a Lightning web component isn't supported.

Use Your Component on a Custom Tab

Alternatively, you can surface your Lightning web component via a custom tab instead of a Lightning page. If you use a custom tab, the tab content and layout can't be edited or configured in Lightning App Builder. Also, custom tabs don't automatically adapt for different devices and screen sizes. So make sure that your component follows responsive design guidelines before you make the custom tab available. See Custom Tabs.

Follow Accessible Mobile Design Guidelines

Before you build and test your components on a mobile screen, follow best practices for making your designs accessible.

Mobile characteristics like smaller viewport size and reduced processing power can constrain your design for layout, control mechanisms, and navigation. These constraints impact accessibility for mobile users. For example, imagine touch targets that are too small and cause a user to abandon your app.

The good news is that mobile devices run on similar web technologies to desktop and can usually handle fully featured websites. To make your components accessible on mobile, follow general web design and accessibility best practices. Consider the accessibility guidelines for mobile screens early in your design so there aren't surprises when you deploy.

Use Base Components and Salesforce Lightning Design System (SLDS)

Use base components whenever possible. Base components implement SLDS styling, so they match the look-and-feel of the Salesforce mobile app and have accessible features built in. To style the base component, use design variations, styling hooks, and design tokens. If these techniques don't meet your requirements, use an SLDS component blueprint to build your own component.

Note: Not all base components are mobile-ready. For example, `lightning-tabset` and `lightning-tab` are based on the Tabs SLDS component blueprint. However, the tabs don't stack on mobile to adapt to a more narrow screen. Refer to the component reference for device support details.

Use Mobile-Friendly Fonts, Padding, and Color

To ensure that your mobile app or page design is consistent, use SLDS tokens. Avoid using font sizes smaller than `$font-size-4` (14 px) as they can be hard to read on small screens. Consider these font guidelines.

- - Use `$font-size-4` (14 px) for secondary text, like input labels
  - Use `$font-size-5` (16 px for primary text, like input values
  - Use `$font-size-6` (18 px) for a heading, like for a section of a form
  - Use `$font-size-7` (20 px) for a title, like for a record name

Padding provides spacing around your content. Use a consistent spacing system to keep your app pages neat. We recommend using the SLDS padding utility for a consistent layout throughout your app. Besides padding, use the various SLDS utilities for alignment, margin, text, and many others.

Colors on a mobile screen are especially important because users have to deal with glare and movement. Text and informational icons should have good color contrast. Aim for a color contrast ratio of 4.5:1 or higher for regular-sized text, 3:1 for icons and large text. Large text is 24 px or more, or 18 px for bold text.

Avoid Horizontal Swiping Tabs and Carousels

Content in tabs and carousels is easy to miss and poses accessibility and discoverability issues. If you must use a tab or carousel, keep swiping to a minimum so that users can reach the last item in 3 or 4 steps. To present a high number of items, we recommend presenting your content vertically in the viewport, such as using a list view. A list view enables users to access any item on the page directly. Alternatively, apply a container around a related grouping of information using the `lightning-card` base component to match the look-and-feel of the Salesforce mobile app. If you're using a table to display data, the table should responsively collapse into tile lists on narrow screens. See Displaying Data.

Provide Alternatives to Gestures

Gestures such as tapping or swiping enable users to interact with screen elements. However, users who require keyboard input or run assistive technology can't perform gestures on their mobile screens. Avoid making gestures the only way to access functionality. When

using a gesture, provide a button or other control to trigger an action. You can enable gestures as shortcuts for users who perform a task repeatedly. If you use a custom gesture, include assistive text to describe the behavior.

Make Control Mechanisms Accessible

Some people use screen readers or voice control for navigation and input. To support accessibility, make control mechanisms such as buttons, links, and form fields focusable. Since buttons are used widely as a call to action, it's important to make them user-friendly and accessible. Button text should have good color contrast ratio for the button background, compared to the background the button is placed on. See MDN: Accessibility concerns for buttons.

By default, native interactive elements, such as `button`, `a`, and `input` are focusable and include the required semantics and behavior. If you must use `div` and `span` elements, set their `tabindex` value to 0 so they receive keyboard focus. Base components have built-in accessibility and take care of the tab order. See Handle Focus.

Links take user away from their current view, so the link content should indicate where the link goes. For example, "click here" is not helpful compared to "about our services". To optimize navigating across the Salesforce mobile app, use the navigation service available in the `lightning/navigation` module. The navigation service enables you to navigate to many different page types, like records, list views, and objects. The navigation service provides extensive routing, deep linking, and login redirection. You can use it for app navigation, state changes, and refreshes. See Navigate to Pages, Records, and Lists.

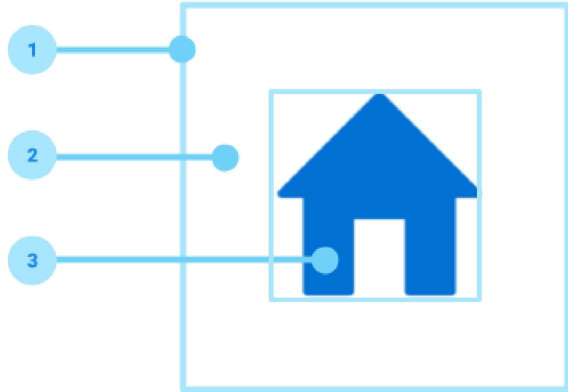
Note: For security reasons, these methods aren't supported: `window.open`, `window.location`, and `location.href`. To navigate to an external URL, use `lightning/navigation` with the `standard__webPage` attribute.

Follow Standard Tap Target Size

Mobile users are constrained with a smaller viewport size. For interactive elements, like buttons and links, provide an area that's large enough to tap or activate the element. It's also helpful to provide spacing to separate interactive elements, since mobile users can easily click or activate the wrong element.

The minimum size for a tap target for any actionable item on mobile devices is 44x44 px. If there is more than one target on a screen that performs the same action, only one of the targets need to meet the target size of 44 by 44 CSS pixels. Secondary tap targets, such as a listview picker or breadcrumb link, can have a minimum size of 32px.

Consider these tap target guidelines.



1. Tap target size: The tap target consists of the visual signifier, the container (if there is one), and the internal or external padding. The minimum size is 44px.

2. Internal padding: Changes based on the size of the visual signifier used.
3. Visual signifier: An avatar, icon, image, or text. The size can change but the tap target cannot go below 44px.

See the WCAG guidelines.

Make User Input as Painless as Possible

Mobile users typically take action or consume information quickly. These quick actions are known as *micro-moments*, short bursts of focused activity that last about a minute on average. If you are requesting user input, minimize the amount of typing for mobile users.

To get user input for Salesforce records, we recommend using the `lightning-record-form` or `lightning-record-edit-form` base component. Alternatively, use `lightning-input` to get user input using the `lightning/ui*Api` Wire Adapters and Functions to work with Salesforce data and metadata. When you use the base components, the `<label>` and `<input>` elements are automatically configured for you to adhere to accessibility best practices. See Work with Records Using Base Components.

Provide a visible label for every form field and avoid long labels or truncating labels on mobile. If you work with long labels, consider placing your labels on the top of the field instead of horizontally aligning them with the field. If you can't include a visible label, provide a hidden label so it's still available to assistive technologies. The base components top aligns the label on the field automatically for you. They also enable you the option to hide a label visually and still make it available to assistive technologies.

SEE ALSO:

Lightning Design System: Accessible Mobile Design Guidelines

MDN: Mobile Accessibility

Lightning Web Components Developer Guide: Component Accessibility

Disable Pull-to-Refresh in the Salesforce Mobile App

Disable pull-to-refresh on pages where accidentally triggering it can cause loss of data in the Salesforce mobile app. Disabling pull-to-refresh is as simple as firing a `CustomEvent`. Fire this event in your own components, or create a component you can use throughout your Salesforce org.

Pull-to-refresh is a long-established convention in mobile apps as a way to reload data appearing on a mobile app screen. It's the default behavior for nearly all screens in the Salesforce mobile app. However, triggering pull-to-refresh while entering data into a form causes the form to refresh, losing values entered into the form. A custom event lets you disable pull-to-refresh on any screen from within a Lightning web component.

First, create a `CustomEvent` with the name `updateScrollSettings`, and a data payload as illustrated here:

```
const disable_ptr_event = new CustomEvent("updateScrollSettings", {
    detail: {
        isPullToRefreshEnabled: false
    },
    bubbles: true,
    composed: true
});
```

Then fire the event:

```
this.dispatchEvent(disable_ptr_event);
```

This event has no effect outside the Salesforce mobile app. You can include it on pages that are shared between desktop and mobile without affecting the behavior of Salesforce for your desktop users.

Example: **Faceless `DisablePullToRefresh` Component**

The following example code shows a component that does only one thing: disable pull-to-refresh on any page that includes it in the Salesforce mobile app. This component is "faceless", in that it doesn't have a user interface, or any visual effect at all on pages that include it.

Create the component in your org, and then use it anywhere you need to disable pull-to-refresh. You can add it to Lightning Pages, flows, and record pages just by adding this component to your page or layout. You can also add it as a child component to any custom component you create where pull-to-refresh could interfere with your component's intended behavior.

Component Metadata

Adding the correct targets to the component metadata allows it to be used in all contexts where it's useful.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- disablePullToRefresh.js-meta.xml -->
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <masterLabel>Disable Pull-to-Refresh (No UI)</masterLabel>
    <description>This component disables "pull to refresh" behavior in the Salesforce
 Mobile app.
    Add it to a page, or as a child component in your component. This component has
no user
    interface, and has no effect outside supported mobile apps.</description>
    <apiVersion>54.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__FlowScreen</target>
        <target>lightning__HomePage</target>
        <target>lightning__RecordAction</target>
        <target>lightning__RecordPage</target>
        <target>lightning__Tab</target>
    </targets>
</LightningComponentBundle>
```

Component Template

This component has no user interface. Its only purpose is to fire the event that disables pull-to-refresh. As such, the component template is empty.

```
<!-- disablePullToRefresh.html -->
<template>
    <!-- This component has no user interface -->
    <!-- It just fires its event, and is done -->
</template>
```

Component Implementation

The component does one thing: fire the event that disables pull-to-refresh as soon as it's loaded. It defines a function that fires the event, and calls that function in the `connectedCallback` lifecycle hook.

```
// disablePullToRefresh.js
import { LightningElement } from 'lwc';

export default class DocDisablePullToRefresh extends LightningElement {
```

```
    // Trigger this component's effect when the component loads
    connectedCallback() {
        this.disablePullToRefresh();
    }

    // Fire the event to disable pull-to-refresh on this page
    // This has an effect only in the Salesforce Mobile and
    // Mobile Publisher apps
    disablePullToRefresh () {
        // CustomEvent is standard JavaScript. See:
        // https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent/CustomEvent
        const disable_ptr_event = new CustomEvent("updateScrollSettings", {
            detail: {
                isPullToRefreshEnabled: false
            },
            bubbles: true,
            composed: true
        });
        this.dispatchEvent(disable_ptr_event);
    }
}
```

SEE ALSO:

Lightning Web Components Developer Guide: XML Configuration File Elements

Lightning Web Components Developer Guide: Lifecycle Hooks

Lightning Web Components Developer Guide: Create and Dispatch Events

CHAPTER 3 Use Mobile Device Features in Mobile Apps

In this chapter ...

- Request an App Review on a Mobile Device
- Scan Barcodes on a Mobile Device
- Access a Mobile Device's Biometrics Capabilities
- Manage Calendar Events on a Mobile Device
- Access Contacts on a Mobile Device
- Scan Documents on a Mobile Device
- Monitor Geofence Regions on a Mobile Device
- Use Location on a Mobile Device
- Interact with NFC Tags on a Mobile Device
- Accept On-Site Payments with Tap-to-Pay

Mobile capabilities let you use mobile device features from within a Lightning web component. Access camera and location detection hardware, and platform features like contacts and calendar data, right from your component code. Build Lightning apps that feel like native mobile apps using these mobile-specific features.

Mobile capabilities are built by Salesforce using the *Nimbus framework*. Nimbus creates a bridge between Lightning web components and a mobile device's native operating system and hardware. *Nimbus plugins* use the Nimbus framework, and are compiled into Salesforce mobile apps, with each plugin providing access to a specific feature area.

Nimbus plugins expose native features to Lightning web components through JavaScript APIs, allowing you to easily access these features in your Lightning web components.

Mobile capabilities built with Nimbus can only be used when your Lightning web component runs in a supported mobile app running on a mobile device. They are built on, and depend on, compiled code included in the mobile app. They **cannot** be used on desktop, or in a mobile web browser.

Mobile Capabilities Compatibility Summary

Mobile capabilities are supported individually by each Salesforce mobile app. Not every mobile capability is supported in every mobile app. The following table provides a compatibility overview, but see the compatibility topic for each mobile capability for full compatibility details.

Mobile Capability	Salesforce Mobile	Salesforce Mobile App Plus	Mobile Publisher	Field Service Mobile
AppReviewService	✗	✗	✓	✗
BarcodeScanner	✓	✓	✓	⚠ See note.
BiometricsService	✓	✓	✓	✗
CalendarService	✗	✓ iOS only.	✓	✗
ContactsService	✗	✓ iOS only.	✓	✗
DocumentScanner	✓	✓	✗	✓
GeofencingService	✗	✗	✗	✓

Mobile Capability	Salesforce Mobile	Salesforce Mobile App Plus	Mobile Publisher	Field Service Mobile
LocationService	✓ Android only.	✓	✓	✓
NFCService	✓	✓	✗	✓ Android only.
PaymentsService	✗	✗	✗	✓

Note: The Field Service Mobile app provides an alternative implementation of BarcodeScanner. See Scan Barcodes on a Mobile Device in the *Field Service Developer Guide* for details.

Request an App Review on a Mobile Device

A Lightning web component can use a mobile device to prompt users to rate and submit a review of your app to the app stores (Apple and Google). Their feedback can help improve your app experience, encourage downloads, and improve your app's discoverability.

AppReviewService requires access to platform-specific APIs that are available only within compatible Salesforce mobile apps.

Important: AppReviewService does not and cannot function when running in a web browser, whether on a desktop or mobile device.

IN THIS SECTION:

AppReviewService User Experience

Your component can deliver any user experience you desire.

Use the AppReviewService API

To develop a Lightning web component with app review features, use the AppReviewService API.

AppReviewService Example

Here's a minimal but complete example of a Lightning web component that uses AppReviewService to request an app review.

Compatibility and Requirements

AppReviewService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when AppReviewService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

Considerations and Limitations

Consider these guidelines and limitations when developing features that use the AppReviewService API.

SEE ALSO:

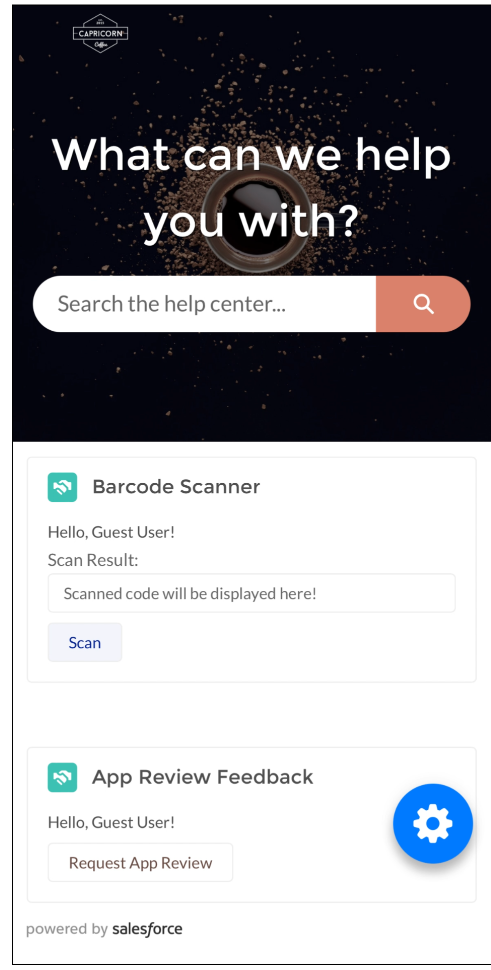
Lightning Web Components Developer Guide: AppReviewService API

AppReviewService User Experience

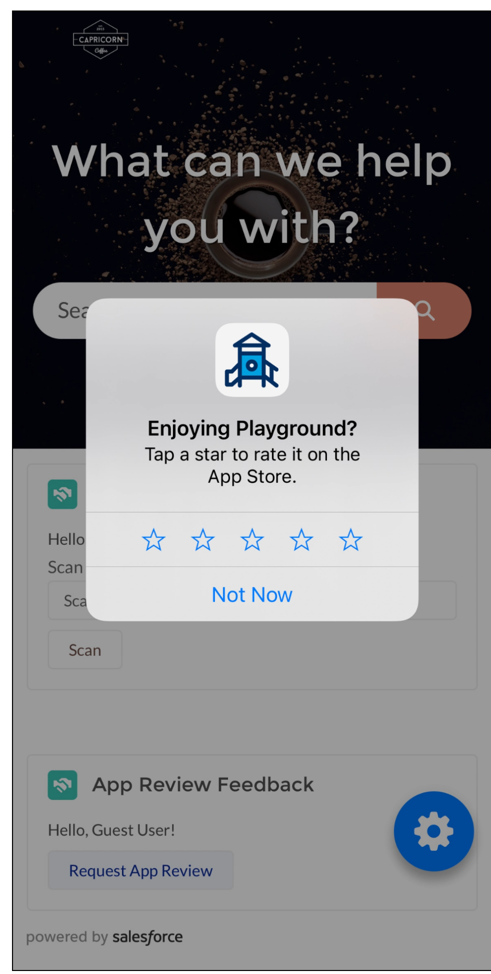
Your component can deliver any user experience you desire.

Here's an example of the app review component.

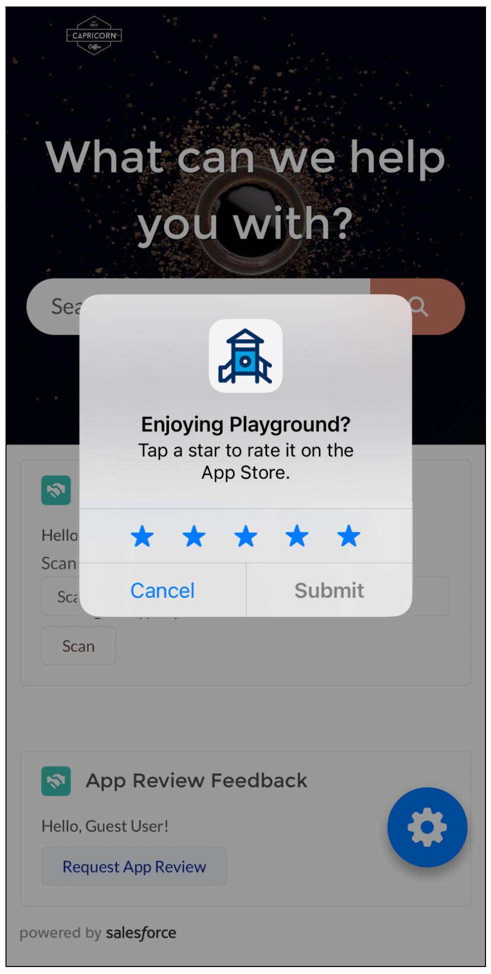
1. A button displays to start the process of requesting an app review.



2. When the button is tapped, AppReviewService opens an app review prompt.



3. After users rate the app, they can submit their rating to the app stores.



Use the AppReviewService API

To develop a Lightning web component with app review features, use the AppReviewService API.

1. Import AppReviewService into your component definition to make the AppReviewService API functions available to your code.
2. Test to make sure AppReviewService is available before you call app feedback functions.
3. Use the app review functions to ask users app review related questions.

Import AppReviewService into a Component

In your component's JavaScript file, import AppReviewService using the standard JavaScript `import` statement. Specifically, import the `getAppReviewService()` factory function from the `lightning/mobileCapabilities` module, like so:

```
import { getAppReviewService } from 'lightning/mobileCapabilities';
```

After it's imported into your component, use the factory function to get an instance of AppReviewService. With your AppReviewService instance, use the utility functions and constants to verify availability. Use the app review related functions to perform app review operations.

Test AppReviewService Availability

AppReviewService depends on physical device hardware and platform features. A component that uses AppReviewService renders without errors on a desktop computer or in a mobile browser, but app review functions fail. To avoid these errors, test if AppReviewService functionality is available before you use it.

```
handleBeginClick(event) {
    const myAppReviewService = getAppReviewService();
    if(myAppReviewService.isAvailable()) {
        // Perform app review related operations
    }
    else {
        // AppReviewService not available
        // Handle with message, error, beep, and so on
    }
}
```

Request an App Review

It's straightforward to create an app review feature using AppReviewService.

1. Start a review request with `requestAppReview()`.
2. Handle the results of the app review request.

For example:

```
myAppReviewService.requestAppReview(null)
.then(() => {
    // Do something with success response
    console.log("App review request complete successfully");
})
.catch((error) => {
    // Handle cancelation and scanning errors here
    console.error(error);
});
```

See `requestAppReview()` for more details about how to handle errors.

SEE ALSO:

Lightning Web Components Developer Guide: AppReviewService API

AppReviewService Example

AppReviewService Example

Here's a minimal but complete example of a Lightning web component that uses AppReviewService to request an app review.

The component's HTML template contains a button to request an app review.

```
<!-- appReviewFeedbackServiceExample.html -->
<template>
    <lightning-card title="App Review Feedback" icon-name="custom:custom14">
        <div class="slds-var-m-around_medium">
            <div>Hello, {name}!</div>
```

```
            <div class="slds-var-m-top_x-small">
                <lightning-button label="Request App Review" value="Action"
onclick={handleBeginClick}></lightning-button>
            </div>
        </div>
    </lightning-card>
</template>
```

Each phase of the app review request writes a console message.

```
// appReviewServiceExample.js
import { LightningElement, wire } from 'lwc';
import { getAppReviewService } from 'lightning/mobileCapabilities';
import { getRecord, getFieldValue } from 'lightning/uiRecordApi';
import Id from '@salesforce/user/Id';
import NAME_FIELD from '@salesforce/schema/User.Name';
const fields = [NAME_FIELD];
const userName = getFieldValue(this.user.data, NAME_FIELD)

export default class AppReviewFeedbackService extends LightningElement {
    userId = Id;
    user;

    @wire(getRecord, { recordId: '$userId', fields })
    user;

    get name() {
        return userName || "Guest User";
    }

    handleBeginClick(event) {
        const myAppReviewService = getAppReviewService();
        if (myAppReviewService.isAvailable()) {
            myAppReviewService.requestAppReview(null)
                .then(() => {
                    // Do something with success response
                    console.log("App review request complete successfully");
                })
                .catch((error) => {
                    // Handle cancellation and scanning errors here
                    console.error(error);
                });
        }
        else {
            // Handle with message, error, beep, and so on
            console.error("App Review service not available");
        }
    }
}
```

SEE ALSO:

Use the AppReviewService API

Compatibility and Requirements

AppReviewService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when AppReviewService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

AppReviewService is available in Lightning apps that are distributed using Mobile Publisher for Experience Cloud.

AppReviewService is fully functional when used in a Lightning app or Lightning site that's run from a compatible Mobile Publisher for Experience Cloud mobile app on a compatible iOS or Android mobile device. See Requirements for Mobile Publisher for Experience Cloud, or the requirements page for your target mobile app for specific device and operating system requirements.

AppReviewService is **not** fully available when running on other devices, such as a desktop, or when running in a web browser, even on a mobile device. It **requires** the Mobile Publisher for Experience Cloud app. The AppReviewService API is accessible in Lightning Experience on all devices, so your code won't fail due to missing functions. However, when your app runs in a browser—desktop or mobile—it can use only AppReviewService constants and utility functions. Attempting any app review-related operation will fail.

Considerations and Limitations

Consider these guidelines and limitations when developing features that use the AppReviewService API.

Development Considerations

You can develop the user experience for your component on a desktop or laptop development system. Be sure to test app review functionality on the physical devices on which you plan to deploy your Lightning app.

Apple and Google provide their own tips and best practices for requesting app reviews.

- iOS: Requesting App Store reviews (Apple)
- iOS: Ratings, Reviews, and Responses
- Android: Google Play In-App Reviews API (Google)

Testing Limitations

iOS

When you test app review features while your app is in development mode, the app rating and review request view is displayed so you can test the UI experience. If you're testing app reviews on an app that you distribute through TestFlight, it won't display the app rating and review request.

Android

You can test the app review integration without publishing your app to production using either internal test tracks or internal app sharing.

- Test using an internal test track (Google)
- Test using internal app sharing (Google)

As you test your app review features on Android, you might encounter issues. To learn about some common issues with reviews on Google Play and their solutions, see Troubleshooting (Google).

Publisher Playground App

While testing app review features isn't available on the Publisher Playground app, you can test it using Playground app virtual device builds.

- Playground App Virtual Device Builds (Salesforce)

Device Limitations

AppReviewService doesn't implement an app review feature itself. Instead, it makes available the native features of the underlying platform (Android or iOS). While the features provided by AppReviewService are the same across both, it's subject to platform-specific quirks and minor differences.

AppReviewService Considerations

Be aware of the following considerations when using AppReviewService in your Lightning app.

- AppReviewService is built on top of mobile operating system and device features. AppReviewService's capabilities therefore depend on Android or iOS features, which are subject to change beyond our control. When mobile operating system features change, the behavior of AppReviewService can change without notice.
- A Lightning component that uses AppReviewService can have a custom user interface in the component itself.

Scan Barcodes on a Mobile Device

A Lightning web component can use a mobile device's camera and mobile OS platform features to scan a barcode, such as a UPC symbol or QR code. When a barcode is successfully scanned, the data that was read from the barcode is returned to the Lightning web component that invoked it.

Scanning is performed locally on the mobile device, and doesn't need a network connection. BarcodeScanner does require access to platform-specific APIs that are available only within compatible Salesforce mobile apps.

Important: BarcodeScanner does not and cannot function when running in a web browser, whether running on a desktop or mobile device.

BarcodeScanner provides to your component a string value of the data encoded in a scanned barcode. It doesn't attempt to interpret or process the decoded value.

IN THIS SECTION:

Barcode Scanning User Experience

Your component can deliver any user experience you desire, but there's a common flow for any component that can scan a barcode.

Use the BarcodeScanner API

To develop a Lightning component with barcode scanning features, use the BarcodeScanner API.

BarcodeScanner Example–Modern Scanning API

Here's a complete example of a Lightning web component that uses BarcodeScanner to scan multiple barcodes simultaneously and process them in a batch after scanning is completed.

BarcodeScanner Example—Single Scan (Legacy)

Here's a minimal but complete example of a Lightning web component that uses BarcodeScanner to recognize a barcode.

Scan Multiple Barcodes (Legacy)

To scan multiple barcodes in a single scanning session, use `resumeCapture()` to create a continuous scanning cycle that scans barcodes until the user clicks the **Cancel** button.

BarcodeScanner Example—Continuous Scanning (Legacy)

Here's a minimal but complete example of a Lightning web component that uses BarcodeScanner to scan for and recognize multiple barcodes in a continuous cycle.

Create a Self-Service Kiosk Application

Use BarcodeScanner with a device's front-facing camera to create applications suitable for use as an unattended self-service kiosk.

BarcodeScanner Example—Self-Service Kiosk (Legacy)

Here's a complete example of a Lightning web component with BarcodeScanner that could serve as a self-service kiosk.

Customize the BarcodeScanner User Interface

BarcodeScanner provides a standard, minimal user interface that can be used out of the box. For applications and use cases where the standard user interface doesn't provide enough information, or to customize for your company or brand, create a custom UI using HTML.

Compatibility and Requirements

BarcodeScanner is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when BarcodeScanner runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a web browser, whether the browser runs on a desktop or mobile device.**

Considerations and Limitations

Keep the following in mind when developing features that use the BarcodeScanner API.

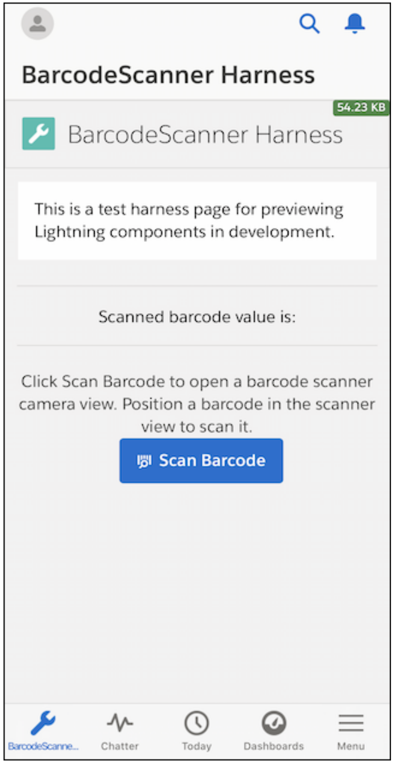
SEE ALSO:

Lightning Web Components Developer Guide: BarcodeScanner API

Barcode Scanning User Experience

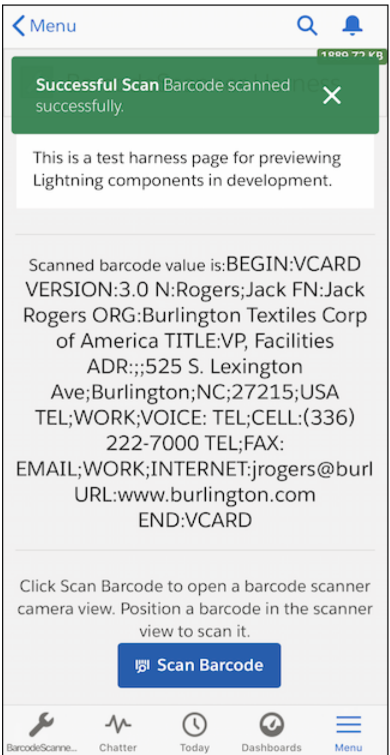
Your component can deliver any user experience you desire, but there's a common flow for any component that can scan a barcode.

1. A Lightning web component displays a button (or other user interface control) to start a scan.



2. When the button is pressed, BarcodeScanner invokes the mobile device's user interface for the camera and barcode scanning function. When the camera detects a valid barcode, it displays a bounding box around the barcode, reads the data from the barcode, and returns the result to the Lightning web component that invoked it.

3. The Lightning web component displays or otherwise processes the results of the scan.



Use the BarcodeScanner API

To develop a Lightning component with barcode scanning features, use the BarcodeScanner API.

1. Import BarcodeScanner into your component definition to make the BarcodeScanner API functions available to your code.
2. Test to make sure BarcodeScanner is available before you call scanning lifecycle functions.
3. Use the scanning lifecycle functions to start, continue, and stop scanning.

Important: We recommend using the modern `scan()` and `dismiss()` API functions in your LWC scanning code to streamline your development experience. The legacy API functions `beginCapture()`, `resumeCapture()`, and `endCapture()` are still available, but will be retired in a future release. See Understand BarcodeScanner Modern and Legacy APIs on page 35 for additional details.

Add BarcodeScanner to a Lightning Web Component

In your component's JavaScript file, import BarcodeScanner using the standard JavaScript `import` statement. Specifically, import the `getBarcodeScanner()` factory function from the `lightning/mobileCapabilities` module, like so:

```
import { getBarcodeScanner } from 'lightning/mobileCapabilities';
```

After it's imported into your component, use the factory function to get an instance of BarcodeScanner. With your BarcodeScanner instance, use the utility functions and constants to verify scanner availability and to configure scans. Use the scanning lifecycle functions to perform scanning operations.

Test BarcodeScanner Availability

BarcodeScanner depends on physical device hardware and platform features. A component that uses BarcodeScanner renders without errors on a desktop computer, but scanning functions fail. To avoid these errors, test if BarcodeScanner functionality is available before you use it.

```
handleBeginScanClick(event) {
    const myScanner = getBarcodeScanner();
    if(myScanner.isAvailable()) {
        // Perform scanning operations
    }
    else {
        // Scanner not available
        // Not running on hardware with a scanner
        // Handle with message, error, beep, and so on
    }
}
```

Scan a Barcode

Scanning with BarcodeScanner is simple using the scanning lifecycle functions.

1. Start a scan with `scan(options)`.
2. Handle the result of the scan, which is returned in the form of a promise.
3. End the scan with `dismiss()`.

For example:

```
myScanner.scan(scanningOptions)
    .then((resultsArray) => {
        // Do something with the result of the scan
        for (let singleResult in resultsArray) {
            console.log(singleResult);
            this.scannedBarcodes.push(singleResult.value);
        }
    })
    .catch((error) => {
        // Handle cancellation and scanning errors here
        console.error(error);
    })
    .finally(() => {
        myScanner.dismiss();
    });
```

See `scan(options)` for more details of how to handle scan results, handle errors, and so on.

IN THIS SECTION:

Understand BarcodeScanner Modern and Legacy APIs

In previous versions of BarcodeScanner, scanning a single barcode in a scanning session required a different programmatic approach than scanning several barcodes in a row without requiring user intervention after each scan. Now, BarcodeScanner has new APIs to streamline the development experience for these common use cases, and new capabilities to scan large quantities of barcodes more efficiently.

Understand the BarcodeScanner Scanning Lifecycle

BarcodeScanner has four distinct scanning modes, each appropriate for different use cases.

SEE ALSO:

Lightning Web Components Developer Guide: BarcodeScanner API

BarcodeScanner Example–Modern Scanning API

BarcodeScanner Example—Single Scan (Legacy)

Understand BarcodeScanner Modern and Legacy APIs

In previous versions of BarcodeScanner, scanning a single barcode in a scanning session required a different programmatic approach than scanning several barcodes in a row without requiring user intervention after each scan. Now, BarcodeScanner has new APIs to streamline the development experience for these common use cases, and new capabilities to scan large quantities of barcodes more efficiently.

Legacy APIs and Modern APIs

We use the terms *legacy APIs* and *modern APIs* here. Let's clarify what they mean.

- **Legacy APIs** refer to the functions `beginCapture()`, `resumeCapture()`, and `endCapture()`
- **Modern APIs** refer to the functions `scan()` and `dismiss()`

The legacy APIs are supported, but will be retired in a future release–the modern APIs replace them fully. The legacy APIs support single scanning and continuous scanning modes, but not bulk scanning or multi-scanning.

The modern APIs are, as you've probably guessed, a newer addition to BarcodeScanner. They were created to simplify the development experience, and they support all scanning modes, including bulk scanning and multi-scanning.

Note: If you're adding BarcodeScanner to your LWC for the first time, use the modern APIs. There's no advantage to using the legacy APIs, and you'll eventually have to switch to the modern APIs anyway, when the retirement of the legacy APIs becomes official.

If you have an existing LWC that uses the legacy APIs, we encourage you to update your code to use the modern APIs as soon as possible, so you can enjoy a more streamlined development experience and also have access to the new bulk scanning and multi-scanning capabilities.

The following table summarizes the relationships of the legacy APIs to the modern APIs replacing them:

Legacy API	Modern API	Notes
`beginCapture()`	`scan()`	`scan()` replaces both `beginCapture()` and `resumeCapture()`.
`resumeCapture()`	`scan()`	
`endCapture()`	`dismiss()`	

Practical Differences Between `scan()` and `beginCapture()`

For the most part, the behavior of the modern APIs is identical to their legacy counterparts. One notable difference is how the returned promise is resolved in `scan()`, compared to `beginCapture()`.

In `beginCapture()`, the returned Promise resolves to **a single result**. In `scan()`, the returned Promise resolves to **an array of results**. Because bulk scanning and multi-scanning process multiple barcodes simultaneously, only the modern `scan()` API supports them.

SEE ALSO:

Lightning Web Components Developer Guide: BarcodeScanner API

Understand the BarcodeScanner Scanning Lifecycle

Use the BarcodeScanner API

Understand the BarcodeScanner Scanning Lifecycle

BarcodeScanner has four distinct scanning modes, each appropriate for different use cases.

The different scanning cycles are the following:

- Single Scanning
- Continuous Scanning
- Bulk Scanning
- Multi-Scanning

Single Scanning

Single scanning mode consists of scanning a single barcode, followed immediately by processing the barcode data.

Single scanning is ideal when the use case is a situation where user interaction is desired right after the barcode is scanned. For example, if your application seeks to confirm the user's identity, one possible implementation starts with a barcode scan and then immediately prompts the user to answer a security question before the next user can scan their barcode.

Continuous Scanning

Continuous scanning mode consists of scanning several barcodes, one after the other, processing each one after it's scanned.

Continuous scanning is ideal where only one barcode must be scanned at a time and the processing must take place after each scan. For example, if your application relates to inventory management in a warehouse, one possible implementation involves employees scanning many items, one at a time, updating the inventory system in real-time and prompting inventory managers to reorder more of a given product if stock runs low.

Bulk Scanning and Multi-Scanning

The default continuous scanning mode (explained in the previous section) is ideal for use cases where you want to scan one barcode at a time, over and over again, processing each barcode as it's scanned. But what if you want to scan several barcodes before processing them? Or scan a bunch simultaneously? Enter **bulk scanning** and **multi-scanning**.

Bulk scanning and multi-scanning are represented as Boolean parameters (`enableBulkScan` and `enableMultiScan`, respectively) on the `BarcodeScannerOptions` object. They're enabled when their values are set to `true`.

Bulk scanning mode consists of scanning several barcodes, one after the other, and then processing all of them in a batch after scanning is completed.

Bulk scanning is ideal when multiple barcodes are to be scanned in a single session, but the processing can take place in one batch at the end, after all items are scanned. For example, if your application is used for checking out materials at a library, one possible implementation involves librarians scanning many books, one at a time. After all the books have been scanned for check-out, the system processes all the books at once, assigning them to the patron checking out the books and marking those books unavailable for other patrons until they're returned.

Multi-scanning mode consists of scanning several barcodes simultaneously, processing all of them in a batch after scanning is completed.

Multi-scanning is ideal when the use case is a situation where many barcodes must be scanned at a time. For example, if your application is used in a manufacturing quality control setting, one possible implementation involves multiple barcodes being placed on each product, at different stages of the manufacturing process. Scanning all of these barcodes simultaneously allows for an efficient quality check, confirming that each product is ready for sale and shipment.

Similarities Between the Scanning Modes

From a technical perspective, there aren't **really** four distinct scanning modes–there are just two.

We've defined four modes in the preceding section because, from a user perspective, those four styles of scanners are what you have to choose from, depending on your use case. However, when it comes to controlling the scanning cycle from your LWC code, single scanning is just a special case of continuous scanning. And, although bulk scanning and multi-scanning provide a different user experience, they're practically identical in the scanning code you need to write. So, single and continuous scanning represent one of the fundamental scanning lifecycles, while bulk and multi-scanning represent the other.

The main difference between these two types of scanning lifecycles is when the processing of the barcode data occurs. For single scanning and continuous scanning, the processing takes place immediately after each scan, while for bulk scanning and multi-scanning it takes place after the user manually ends the scanning cycle. **Your** code handles the actual processing of the barcode data, so be sure to select the type of scanning behavior that works best for your application.

BarcodeScanner Example–Modern Scanning API

Here's a complete example of a Lightning web component that uses BarcodeScanner to scan multiple barcodes simultaneously and process them in a batch after scanning is completed.

The HTML template provides a minimal scanning user interface. There's an element to display the results of the scans in a list view, a bit of static help text, and a button to start scanning.

```
    <!-- barcodeScannerMultiScan.html -->
  <template>
    <div class="slds-text-align_center">
      <span class="slds-text-heading_large">BarcodeScanner: Multi-Scan</span>
    </div>

   <!-- Static help text -->
  <div class="slds-text-color_weak slds-m-vertical_large slds-m-horizontal_medium">
    <p>Tap <strong>Start a Scanning Session</strong> to open a barcode scanner camera view.
 Position barcodes in the scanner view to scan them.</p>
    <p>Continue scanning items. Tap Done when you are done scanning.</p>
  </div>

   <!-- Scan button, always enabled -->
  <div class="slds-align_absolute-center slds-m-vertical_large">
    <lightning-button
     variant="brand"
     class="slds-var-m-left_x-small"
     icon-name="utility:scan"
     label="Start a Scanning Session"
     title="Start scanning barcodes, until there are no more barcodes to scan"
     onclick={beginScanning}
    ></lightning-button>
  </div>

   <!-- After barcodes are scanned, their values are displayed here: -->
    <template lwc:if={scannedBarcodes}>
      <div class="slds-var-m-vertical_large slds-var-p-vertical_medium slds-border_top
slds-border_bottom">
        <p>Scanned barcode values are:</p>
        <pre>{scannedBarcodesAsString}</pre>
      </div>
    </template>
  </template>
```

This example displays all the values of successful scans in a list view. It's a streamlined example, emphasizing the scanning lifecycle of the modern scanning APIs.

```
// barcodeScannerMultiScan.js
import { LightningElement, track } from "lwc";
import { getBarcodeScanner } from "lightning/mobileCapabilities";

export default class BarcodeScannerContinuousDocDemo extends LightningElement {
  barcodeScanner;
```

```
  @track scannedBarcodes;

  connectedCallback() {
    this.barcodeScanner = getBarcodeScanner();
  }

  beginScanning() {
    // Set your configuration options, including bulk and multi-scanning if desired, in
this scanningOptions object
    const scanningOptions = {
      barcodeTypes: [this.barcodeScanner.barcodeTypes.QR],
      scannerSize: "FULLSCREEN",
      cameraFacing: "BACK",
      showSuccessCheckMark: true,
      enableBulkScan: true,
      enableMultiScan: true,
    };

    // Make sure BarcodeScanner is available before trying to use it
    if (this.barcodeScanner != null && this.barcodeScanner.isAvailable()) {
      // Reset scannedBarcodes before starting new scanning session
      this.scannedBarcodes = [];

      // Start scanning barcodes
      this.barcodeScanner
        .scan(scanningOptions)
        .then((results) => {
          this.processScannedBarcodes(results);
        })
        .catch((error) => {
          this.processError(error);
        })
        .finally(() => {
          this.barcodeScanner.dismiss();
        });
    } else {
      console.log("BarcodeScanner unavailable. Non-mobile device?");
    }
  }

  processScannedBarcodes(barcodes) {
    // Do something with the barcode scan value:
    // - look up a record
    // - create or update a record
    // - parse data and put values into a form
    // - and so on; this is YOUR code
    console.log(JSON.stringify(barcodes));
    this.scannedBarcodes = this.scannedBarcodes.concat(barcodes);
  }

  processError(error) {
    // Check to see if user ended scanning
    if (error.code == "USER_DISMISSED") {
      console.log("User terminated scanning session.");
```

```
    } else {
      console.error(error);
    }
  }

  get scannedBarcodesAsString() {
    return this.scannedBarcodes.map((barcode) => barcode.value).join("\n");
  }
}
```

SEE ALSO:

Use the BarcodeScanner API

BarcodeScanner Example—Single Scan (Legacy)

Here's a minimal but complete example of a Lightning web component that uses BarcodeScanner to recognize a barcode.

Important: We recommend using the modern `scan()` and `dismiss()` API functions in your LWC scanning code to streamline your development experience. The legacy API functions `beginCapture()`, `resumeCapture()`, and `endCapture()` are still available, but will be retired in a future release. See Understand BarcodeScanner Modern and Legacy APIs on page 35 for additional details.

The HTML template provides the bare minimum for a scanning user interface. There's an element to display the results of a scan, a bit of static help text, and a button to start a scan. The only thing mildly interesting is the use of the `disabled` attribute to disable the scan button when not on a mobile device. This attribute is set based on the results of `isAvailable()` when the component is initialized.

```
<!-- barcodeScannerExample.html -->
<template>
    <div class="slds-text-align_center">
        <span class="slds-text-heading_large">BarcodeScanner: Single Scan</span>
    </div>

    <!-- After a barcode is successfully scanned,
         its value is displayed here: -->
    <template lwc:if={scannedBarcode}>
        <div class="slds-var-m-vertical_large slds-var-p-vertical_medium
             slds-text-align_center slds-border_top slds-border_bottom">
            Scanned barcode value is:
            <span class="slds-text-heading_small">{scannedBarcode}</span>
        </div>
    </template>

    <!-- Static help text -->
    <div class="slds-text-align_center slds-text-color_weak slds-m-vertical_large">
       Click <strong>Scan Barcode</strong> to open a barcode scanner camera view. Position
 a
        barcode in the scanner view to scan it.
    </div>

    <!-- The click-to-scan button;
         Disabled if BarcodeScanner isn't available -->
```

```
    <div class="slds-align_absolute-center slds-m-vertical_large">
        <lightning-button
            variant="brand"
            class="slds-var-m-left_x-small"
            disabled={scanButtonDisabled}
            icon-name="utility:scan"
            label="Scan Barcode"
            title="Open a camera view and look for a barcode to scan"
            onclick={handleBeginScanClick}>
        </lightning-button>
    </div>
</template>
```

This simple example displays the decoded value of a successful scan. It also displays a toast-style message based on the results of the scan. Each phase of the scanning lifecycle writes a console message.

```
// barcodeScannerExample.js
import { LightningElement } from 'lwc';
import { ShowToastEvent } from 'lightning/platformShowToastEvent';
import { getBarcodeScanner } from 'lightning/mobileCapabilities';

export default class BarcodeScannerExample extends LightningElement {
    myScanner;
    scanButtonDisabled = false;
    scannedBarcode = '';

    // When component is initialized, detect whether to enable Scan button
    connectedCallback() {
        this.myScanner = getBarcodeScanner();
        if (this.myScanner == null || !this.myScanner.isAvailable()) {
            this.scanButtonDisabled = true;
        }
    }

    handleBeginScanClick(event) {
        // Reset scannedBarcode to empty string before starting new scan
        this.scannedBarcode = '';

        // Make sure BarcodeScanner is available before trying to use it
        // Note: We _also_ disable the Scan button if there's no BarcodeScanner
        if (this.myScanner != null && this.myScanner.isAvailable()) {
            const scanningOptions = {
                barcodeTypes: [this.myScanner.barcodeTypes.QR],
                instructionText: 'Scan a QR Code',
                successText: 'Scanning complete.'
            };
            this.myScanner
                .beginCapture(scanningOptions)
                .then((result) => {
                    console.log(result);

                    // Do something with the barcode scan value:
                    // - look up a record
                    // - create or update a record
                    // - parse data and put values into a form
```

```
                    // - and so on; this is YOUR code
                    // Here, we just display the scanned value in the UI
                    this.scannedBarcode = result.value;
                    this.dispatchEvent(
                        new ShowToastEvent({
                            title: 'Successful Scan',
                            message: 'Barcode scanned successfully.',
                            variant: 'success'
                        })
                    );
                })
                .catch((error) => {
                    // Handle cancellation and unexpected errors here
                    console.error(error);

                    if (error.code == 'userDismissedScanner') {
                        // User clicked Cancel
                        this.dispatchEvent(
                            new ShowToastEvent({
                                title: 'Scanning Cancelled',
                                message:
                                    'You cancelled the scanning session.',
                                mode: 'sticky'
                            })
                        );
                    }
                    else {
                        // Inform the user we ran into something unexpected
                        this.dispatchEvent(
                            new ShowToastEvent({
                                title: 'Barcode Scanner Error',
                                message:
                                    'There was a problem scanning the barcode: ' +
                                    error.message,
                                variant: 'error',
                                mode: 'sticky'
                            })
                        );
                    }
                })
                .finally(() => {
                    console.log('#finally');

                    // Clean up by ending capture,
                    // whether we completed successfully or had an error
                    this.myScanner.endCapture();
                });
        } else {
            // BarcodeScanner is not available
            // Not running on hardware with a camera, or some other context issue
            console.log(
                'Scan Barcode button should be disabled and unclickable.'
            );
            console.log('Somehow it got clicked: ');
```

```
            console.log(event);

            // Let user know they need to use a mobile phone with a camera
            this.dispatchEvent(
                new ShowToastEvent({
                    title: 'Barcode Scanner Is Not Available',
                    message:
                        'Try again from the Salesforce app on a mobile device.',
                    variant: 'error'
                })
            );
        }
    }
}
```

SEE ALSO:

BarcodeScanner Example–Modern Scanning API

Use the BarcodeScanner API

Scan Multiple Barcodes (Legacy)

To scan multiple barcodes in a single scanning session, use `resumeCapture()` to create a continuous scanning cycle that scans barcodes until the user clicks the **Cancel** button.

Important: We recommend using the modern `scan()` and `dismiss()` API functions in your LWC scanning code to streamline your development experience. The legacy API functions `beginCapture()`, `resumeCapture()`, and `endCapture()` are still available, but will be retired in a future release. See Understand BarcodeScanner Modern and Legacy APIs on page 35 for additional details.

Sometimes you want to scan many barcodes in a row, without requiring user interaction between scans. For example, when scanning a shelf of inventory, you might not want to stop after each item, or to click a **Scan** button for every item. In these cases, it can make more sense to click **Scan** once, and then scan barcodes repeatedly until done with all of the items. Implementing a continuous scanning cycle like this is slightly different from scanning a single item.

1. Start a scanning session as usual, with `beginCapture()`.
2. When the promise resolves, process the scanned barcode as usual, in the `then()` block.

   Note: Processing the barcode can't change the user interface, or require interacting with the user. That needs to wait until *after* the scanning cycle completes.

3. At the end of the `then()` block, call a new continue scanning function, which uses `resumeCapture()` to continue the current scanning session.
4. Call `endCapture()` at the end of the `catch()` block, instead of in the `finally()` block.
5. When the user clicks **Cancel** to end the scanning session, `BarcodeScanner` returns a `BarcodeScannerError` object with a `code` property value of `userDismissedScanner`. Handle cancellation and actual errors in the `catch()` block.

Single Scan vs. Continuous Scan

The core scanning lifecycles for single scans and continuous scanning are similar, but different enough that it's worth comparing the two.

<table>
<tr><th>Single Scan</th><th>Continuous Scan</th></tr>
<tr><td>

```
singleScanner.beginCapture(scanningOptions)

    .then((scannedBarcode) =>

this.processScannedBarcode(scannedBarcode)

    )
    .catch((error) => {
        console.error(error);

    })
    .finally(() => {
        singleScanner.endCapture();
    });
```

</td><td>

```
sessionScanner.beginCapture(scanningOptions)

    .then((scannedBarcode) =>

this.processScannedBarcode(scannedBarcode)
        this.continueScanning();
    )
    .catch((error) => {
        console.error(error);
        sessionScanner.endCapture();
    });
```

</td></tr>
</table>

There are two significant differences to note.

- The continuous scanning `then()` block has a call to a new function, `continueScanning()`. See the **Continue a Scanning Session** section.
- The call to `endCapture()` is made in the `finally()` block for the single scan, but is called in the `catch()` error handling block for continuous scanning. See the **End Capture for Continuous Scanning** section.

Continue a Scanning Session

In the preceding code comparison, the technique of continuing a scanning session was hidden behind the new line, a call to `continueScanning()`. Here's an example of that function.

```
continueScanning() {
    this.sessionScanner.resumeCapture()
    .then((scannedBarcode) => {
        this.processScannedBarcode(scannedBarcode);
        this.continueScanning();
    })
    .catch((error) => {
        this.processError(error);
        this.sessionScanner.endCapture();
    });
}
```

This code should look familiar; it's nearly identical to the earlier `beginCapture()` example for continuous scanning. There's only one difference: `continueScanning()` creates a promise chain by calling `sessionScanner.`**`resume`**`Capture()`, while the earlier example called `sessionScanner.`**`begin`**`Capture()`. It might be obvious, but you only call `beginCapture()` *once,* at the beginning of a scanning cycle.

While the difference in the code is minor, the difference in the flow of execution is significant. The scanning cycle begins in a promise chain created by `beginCapture()`, which executes **only once.** That initial promise resolves one of two ways:

- If the scan is successful, the barcode result is processed, and then the flow of control for the scanning cycle is handed off to `continueScanning()`.

- If the user clicks **Cancel**, or if there's an error, the promise chain ends in the `catch()` error handling block, covered in the **End Capture for Continuous Scanning** section.

The promise chain in `continueScanning()` ends the same two ways, with one important difference. While the code is the same, after a successful scan it continues the scanning cycle by calling *itself*, creating a recursive loop that continues the scanning cycle until the user clicks **Cancel**, or there's an error.

Whoops. Didn't mean to scare you with that word, *recursive*. Yes, `continueScanning()` ends by calling itself, which makes it a recursive function. But this recursion is pretty simple—it's just a loop, an event loop of sorts. The loop handles *scan-something* events (in the `then()` block) until a *user-clicked-Cancel* event comes along (in the `catch()` block), and then it ends. It might take a minute, but you can wrap your head around it.

The overall pattern here is the following:

- You begin a scanning cycle using `beginCapture()`.
- The promise resolution chain from `beginCapture()` ends in a call to `continueScanning()`, your own function.
- `continueScanning()` continues the existing scanning cycle by calling `resumeCapture()`, but is otherwise the same as the `beginCapture()` that started the cycle.
- The promise resolution chain in `continueScanning()` ends in a call to `continueScanning()`, creating a scanning cycle loop.
- The loop ends when the user clicks **Cancel**, `BarcodeScanner` rejects the promise with a `BarcodeScannerError`, and you call `endCapture()` in the error handling `catch()` block.

The code duplication between the `beginCapture()` and `resumeCapture()` promise chains is unfortunate, but unavoidable. Move as much processing code, such as the handling of a scanned barcode, into functions you can call from both chains. In the example here, `processScannedBarcode()` is a function that both promise chains use to handle a successful scan. See BarcodeScanner Example—Continuous Scanning (Legacy) on page 45 for the complete sample, which includes that function's implementation.

End Capture for Continuous Scanning

In a continuous scanning session, the user scans items repeatedly until they're out of items. Then they click the **Cancel** button to end the session. In code, `BarcodeScanner` handles cancellation by rejecting the promise, and returning a `BarcodeScannerError` to signal that the user canceled scanning. See BarcodeScanner Example—Continuous Scanning (Legacy) on page 45 for how to distinguish between the user clicking **Cancel** and an actual error.

Importantly, clicking the **Cancel** button is the only way to end a continuous scanning session. This is in contrast to a single scan session, which can end with either a successful scan *or* the **Cancel** button.

Because continuous scanning always ends with the **Cancel** button, and thus a `BarcodeScannerError`, we can call `endCapture()` in the error handling `catch()` block.

However, because a single scan might *not* end in a `BarcodeScannerError`, the `catch()` block might never execute. So for a single scan, we put `endCapture()` in the `finally()` block, to make sure that, success or failure, it always gets called.

SEE ALSO:

BarcodeScanner Example—Continuous Scanning (Legacy)

BarcodeScanner Example—Continuous Scanning (Legacy)

Here's a minimal but complete example of a Lightning web component that uses BarcodeScanner to scan for and recognize multiple barcodes in a continuous cycle.

Important: We recommend using the modern `scan()` and `dismiss()` API functions in your LWC scanning code to streamline your development experience. The legacy API functions `beginCapture()`, `resumeCapture()`, and `endCapture()` are still available, but will be retired in a future release. See Understand BarcodeScanner Modern and Legacy APIs on page 35 for additional details.

The HTML template provides the bare minimum for a scanning user interface. There's an element to display the results of the scans, a bit of static help text, and a button to start scanning.

```
<!-- barcodeScannerContinuous.html -->
<template>
    <div class="slds-text-align_center">
        <span class="slds-text-heading_large">BarcodeScanner: Multi-Scan</span>
    </div>

    <!-- After barcode are scanned, their values are displayed here: -->
    <template lwc:if={scannedBarcodes}>
        <div class="slds-var-m-vertical_large slds-var-p-vertical_medium
            slds-text-align_center slds-border_top slds-border_bottom">
            Scanned barcode values are:
            <span class="slds-text-heading_small">{scannedBarcodesAsString}</span>
        </div>
    </template>

    <!-- Static help text -->
    <div class="slds-text-align_center slds-text-color_weak slds-m-vertical_large">
        Click <strong>Start a Scanning Session</strong> to open a
        barcode scanner camera view. Position a barcode in the scanner
        view to scan it.

        <p>Continue scanning items. Click □ when there are no
           more items to scan.</p>
    </div>

    <!-- Scan button, always enabled -->
    <div class="slds-align_absolute-center slds-m-vertical_large">
        <lightning-button
            variant="brand"
            class="slds-var-m-left_x-small"
            icon-name="utility:scan"
            label="Start a Scanning Session"
            title="Start scanning barcodes, until there are no more barcodes to scan"
            onclick={beginScanning}
        ></lightning-button>
    </div>
</template>
```

This example displays all of the values of successful scans, one after the other. This example is streamlined, omitting some of the comments and processing illustrated in BarcodeScanner Example—Single Scan (Legacy) on page 40, to focus on the scanning cycle itself.

```
// barcodeScannerContinuous.js
import { LightningElement, track } from 'lwc';
import { ShowToastEvent } from 'lightning/platformShowToastEvent';
import { getBarcodeScanner } from 'lightning/mobileCapabilities';

export default class BarcodeScannerContinuous extends LightningElement {
```

```
sessionScanner;
@track scannedBarcodes;

connectedCallback() {
    this.sessionScanner = getBarcodeScanner();
}

beginScanning() {
    // Reset scannedBarcodes before starting new scanning session
    this.scannedBarcodes = [];

    // Make sure BarcodeScanner is available before trying to use it
    if (this.sessionScanner != null && this.sessionScanner.isAvailable()) {
        const scanningOptions = {
            barcodeTypes: [this.sessionScanner.barcodeTypes.QR],
            instructionText: 'Scan barcodes — Click □□ when done',
            successText: 'Successful scan.'
        };
        this.sessionScanner.beginCapture(scanningOptions)
        .then((scannedBarcode) => {
            this.processScannedBarcode(scannedBarcode);
            this.continueScanning();
        })
        .catch((error) => {
            this.processError(error);
            this.sessionScanner.endCapture();
        })
    }
    else {
        console.log("BarcodeScanner unavailable. Non-mobile device?");
    }
}

async continueScanning() {
    // Pretend to do some work; see timing note below.
    await new Promise((resolve) => setTimeout(resolve, 1000));

    this.sessionScanner.resumeCapture()
    .then((scannedBarcode) => {
        this.processScannedBarcode(scannedBarcode);
        this.continueScanning();
    })
    .catch((error) => {
        this.processError(error);
        this.sessionScanner.endCapture();
    })
}

processScannedBarcode(barcode) {
    // Do something with the barcode scan value:
    // - look up a record
    // - create or update a record
    // - parse data and put values into a form
    // - and so on; this is YOUR code
```

```
        console.log(JSON.stringify(barcode));
        this.scannedBarcodes.push(barcode);
    }

    processError(error) {
        // Check to see if user ended scanning
        if (error.code == 'userDismissedScanner') {
            console.log('User terminated scanning session via Cancel.');
        }
        else {
            console.error(error);
        }
    }

    get scannedBarcodesAsString() {
        return this.scannedBarcodes.map(barcodeResult => {
            return barcodeResult.value;
        }).join('\n\n');
    }
}
```

Note: This example doesn't process a scanned barcode in any meaningful way. As a result, the `processScannedBarcode()` function executes quickly—*too* quickly. It can trigger a timing issue that causes the example to fail. To avoid the issue, we've inserted a one-second delay before starting the next scan. Real-world barcode processing typically takes long enough to avoid the issue. In that case, you can remove the line with the delay and the `async` keyword preceding the `continueScanning()` function.

See Scan Multiple Barcodes (Legacy) on page 43 for an explanation of how `beginScanning()` and `continueScanning()` work together to create the continuous scanning cycle.

SEE ALSO:

Scan Multiple Barcodes (Legacy)

Create a Self-Service Kiosk Application

Use BarcodeScanner with a device's front-facing camera to create applications suitable for use as an unattended self-service kiosk.

Imagine a convention hall where convention attendees scan their badge for entry. The convention organizers have set up an entry point with a row of tablets, and on each tablet screen is a check-in application, displaying both a scanner view and instructions prompting attendees to scan their badge. Attendees can easily hold their badge up to the scanner view, after which they receive a confirmation message (indicating that they may enter) or a warning message (indicating that there is an issue with their badge, and that they should head over to the information booth to get help).

At the same conference, presenters at each booth in the convention hall need a way to collect leads and stay connected with conference attendees who express interest in their product or service. Each booth is outfitted with a tablet with a scanner view, similar to what the attendees encountered when entering the conference. These scanners, however, scan an attendee's badge for the purpose of collecting their information and sharing it with the booth presenter, so that after the conference the two parties can stay connected, finalize purchase discussions, and more.

To implement these self-service scanning stations, conference organizers and booth presenters used BarcodeScanner in their applications to seamlessly integrate their check-in and lead collection processes with Salesforce. Once scanned, the check-in application processes the barcode to find the associated registration record and verify that the person scanning the barcode is a registered conference attendee. Along the same lines, the booth scanner scans barcodes, extracts embedded attendee data, and uses it to create leads in Salesforce.

Whether you want to update an existing LWC component to use the self-service features of BarcodeScanner, or you're creating a brand-new component, you only need to keep a few things in mind to get started.

What You'll Need

As with all mobile capabilities, the user interface and other implementation details for creating a self-service "kiosk" application are up to you, the LWC developer, to create and maintain. However, there are common elements that any implementation will have.

Important: We recommend using the modern `scan()` and `dismiss()` API functions in your LWC scanning code to streamline your development experience. The legacy API functions `beginCapture()`, `resumeCapture()`, and `endCapture()` are still available, but will be retired in a future release. See Understand BarcodeScanner Modern and Legacy APIs on page 35 for additional details.

Continuous Scanning Lifecycle

To create a scanner that continuously scans and processes barcode data without manual user intervention, use the `resumeCapture()` function in your programming logic to create a continuous scanning lifecycle.

For more information about `resumeCapture()` and how the scanning lifecycle works under the hood, see Scan Multiple Barcodes on page 43.

Front-Facing Camera

BarcodeScanner functionality to use the front-facing camera allows for a better user experience when creating a kiosk application. It allows the screen of your kiosk setup to function as a sort of mirror, which helps your users more easily position their scannable code (whether the code is on a badge, card, or something else) within the scanner view. To use the front-facing camera in your component, set the value of the `cameraFacing` property on the `BarcodeScannerOptions` object to `FRONT`.

For more information on this property and other details of configuring BarcodeScanner, see BarcodeScanner Data Types.

Custom Scanner UI

Finally, you'll want to add your own custom UI to BarcodeScanner to replace the standard, minimal UI. To do this, first build your user interface as custom, static HTML page. Then provide the HTML for your custom UI as a string for the value of the `backgroundViewHTML` property of the `BarcodeScannerOptions` object.

Note: Your custom UI **completely** replaces the standard BarcodeScanner UI, including the **Cancel** button used for dismissing the scanner. Be sure to include this essential element in your custom UI, as well as any other user interface details, such as custom graphics or instructions, you want for your component.

For more information on customizing the UI, see Customize the BarcodeScanner User Interface on page 54.

Putting It All Together

With all of these pieces in place, you'll have an LWC that can serve as a kiosk for continuously scanning barcodes (and processing the data in whatever way your component allows for) without the need to interact with the physical device.

Here are examples of both the standard BarcodeScanner UI and a custom UI appropriate for a Kiosk Mode implementation.

Standard UI

Custom UI

SEE ALSO:

BarcodeScanner Example—Self-Service Kiosk (Legacy)

Customize the BarcodeScanner User Interface

BarcodeScanner Example—Self-Service Kiosk (Legacy)

Here's a complete example of a Lightning web component with BarcodeScanner that could serve as a self-service kiosk.

Important: We recommend using the modern `scan()` and `dismiss()` API functions in your LWC scanning code to streamline your development experience. The legacy API functions `beginCapture()`, `resumeCapture()`, and `endCapture()` are still available, but will be retired in a future release. See Understand BarcodeScanner Modern and Legacy APIs on page 35 for additional details.

The HTML template provides the bare minimum for a scanning user interface. There's an element to display the results of the scans, a bit of static help text, and a button to start scanning.

```
<!-- barcodeScannerKiosk.html -->
<template>
    <div class="slds-text-align_center">
        <span class="slds-text-heading_large">BarcodeScanner: Multi-Scan</span>
    </div>

    <!-- After barcode are scanned, their values are displayed here: -->
    <template lwc:if={scannedBarcodes}>
        <div class="slds-var-m-vertical_large slds-var-p-vertical_medium
            slds-text-align_center slds-border_top slds-border_bottom">
            Scanned barcode values are:
            <span class="slds-text-heading_small">{scannedBarcodesAsString}</span>
        </div>
    </template>

    <!-- Static help text -->
    <div class="slds-text-align_center slds-text-color_weak slds-m-vertical_large">
        Click <strong>Start a Scanning Session</strong> to open a
        barcode scanner camera view. Position a barcode in the scanner
        view to scan it.

        <p>Continue scanning items. Click □ when there are no
           more items to scan.</p>
    </div>

    <!-- Scan button, always enabled -->
    <div class="slds-align_absolute-center slds-m-vertical_large">
        <lightning-button
            variant="brand"
            class="slds-var-m-left_x-small"
            icon-name="utility:scan"
            label="Start a Scanning Session"
            title="Start scanning barcodes, until there are no more barcodes to scan"
            onclick={beginScanning}
        ></lightning-button>
```

```
    </div>

    <!-- Custom UI for the scanner is defined here. We set display:none here because the
scanner will show this. -->
    <div data-id="BarcodeScannerCustomUI" style="display: none;">
        <div>
        <h1 align="right"><a style="text-decoration: none;"
href="nimbusbarcodescanner://dismiss">□</a></h1>
        <h2 align="center">Welcome, let's get you verified!</h2>
        <h3 align="center">Point the front side of your Health Card<br>at the camera on
this device.</h3>
        </div>
    </div>
</template>
```

This example borrows heavily from the code sample in BarcodeScanner Example—Continuous Scanning (Legacy) on page 45. The differences in this example provide all the basic elements needed for a self-service kiosk use case. It uses the front-facing camera for scanning, employs a continuous scanning lifecycle to minimize the need for user interaction, and even defines and uses a custom UI.

```
// barcodeScannerKiosk.js
import { LightningElement, track } from 'lwc';
import { ShowToastEvent } from 'lightning/platformShowToastEvent';
import { getBarcodeScanner } from 'lightning/mobileCapabilities';

export default class NimbusPluginBarcodeScannerCustomUI extends LightningElement {

    sessionScanner;
    @track scannedBarcodes;

    connectedCallback() {
        this.sessionScanner = getBarcodeScanner();
    }

    beginScanning() {
        // Reset scannedBarcodes before starting new scanning session
        this.scannedBarcodes = [];

        // Make sure BarcodeScanner is available before trying to use it
        if (this.sessionScanner != null && this.sessionScanner.isAvailable()) {
           let elem = this.template.querySelector('div[data-id=BarcodeScannerCustomUI]');

            let backgroundViewHTML = '<header><meta name="viewport"
content="width=device-width, initial-scale=1.0, maximum-scale=1.0,
minimum-scale=1.0"></header>';
            backgroundViewHTML += `<html><body>${elem.innerHTML}</body></html>`;

           // Specify the size of the scanner camera view, use of the front-facing camera,
 and pull in the custom UI defined above
            const scanningOptions = {
                "barcodeTypes": [this.sessionScanner.barcodeTypes.QR],
                "scannerSize": "XLARGE",
                "cameraFacing": "FRONT",
                "showSuccessCheckMark": true,
                "presentWithAnimation": false,
                "backgroundViewHTML": backgroundViewHTML
```

```
            };

            this.sessionScanner.beginCapture(scanningOptions)
            .then((scannedBarcode) => {
                this.processScannedBarcode(scannedBarcode);
                this.continueScanning();
            })
            .catch((error) => {
                this.processError(error);
                this.sessionScanner.endCapture();
            })
        }
        else {
            console.log("BarcodeScanner unavailable. Non-mobile device?");
        }
    }

    async continueScanning() {
        // Pretend to do some work; see timing note below.
        await new Promise((resolve) => setTimeout(resolve, 1000));

        this.sessionScanner.resumeCapture()
        .then((scannedBarcode) => {
            this.processScannedBarcode(scannedBarcode);
            this.continueScanning();
        })
        .catch((error) => {
            this.processError(error);
            this.sessionScanner.endCapture();
        })
    }

    processScannedBarcode(barcode) {
        // Do something with the barcode scan value:
        // - look up a record
        // - create or update a record
        // - parse data and put values into a form
        // - and so on; this is YOUR code
        console.log(JSON.stringify(barcode));
        this.scannedBarcodes.push(barcode);
    }

    processError(error) {
        // Check to see if user ended scanning
        if (error.code == 'userDismissedScanner') {
            console.log('User terminated scanning session via Cancel.');
        }
        else {
            console.error(error);
        }
    }

    get scannedBarcodesAsString() {
        return this.scannedBarcodes.map(barcodeResult => {
```

```
            return barcodeResult.value;
        }).join('\n\n');
    }
}
```

Note: This example doesn't process a scanned barcode in any meaningful way. As a result, the `processScannedBarcode()` function executes quickly—*too* quickly. It can trigger a timing issue that causes the example to fail. To avoid the issue, we've inserted a one-second delay before starting the next scan. Real-world barcode processing typically takes long enough to avoid the issue. In that case, you can remove the line with the delay and the `async` keyword preceding the `continueScanning()` function.

See Scan Multiple Barcodes (Legacy) on page 43 for an explanation of how `beginScanning()` and `continueScanning()` work together to create the continuous scanning cycle.

SEE ALSO:

Create a Self-Service Kiosk Application
Customize the BarcodeScanner User Interface

Customize the BarcodeScanner User Interface

BarcodeScanner provides a standard, minimal user interface that can be used out of the box. For applications and use cases where the standard user interface doesn't provide enough information, or to customize for your company or brand, create a custom UI using HTML.

Using HTML to define your custom user interface gives you a lot of flexibility for your UI. Here's an example of the HTML for a minimal custom UI:

```
<header><meta name="viewport" content="width=device-width, initial-scale=1.0,
    maximum-scale=1.0, minimum-scale=1.0"></header>
<html>
  <head>
    <style>
        a:link { text-decoration: none; }
        a:hover { text-decoration: none; }
    </style>
  </head>
  <body>
    <h1 align="right"><a href="nimbusbarcodescanner://dismiss">□</a></h1>
    <h2 align="center">Welcome, let's get you verified!</h2>
    <h3 align="center">Point the front side of your Health Card<br/>
      at the camera on this device.</h3>
  </body>
<html>
```

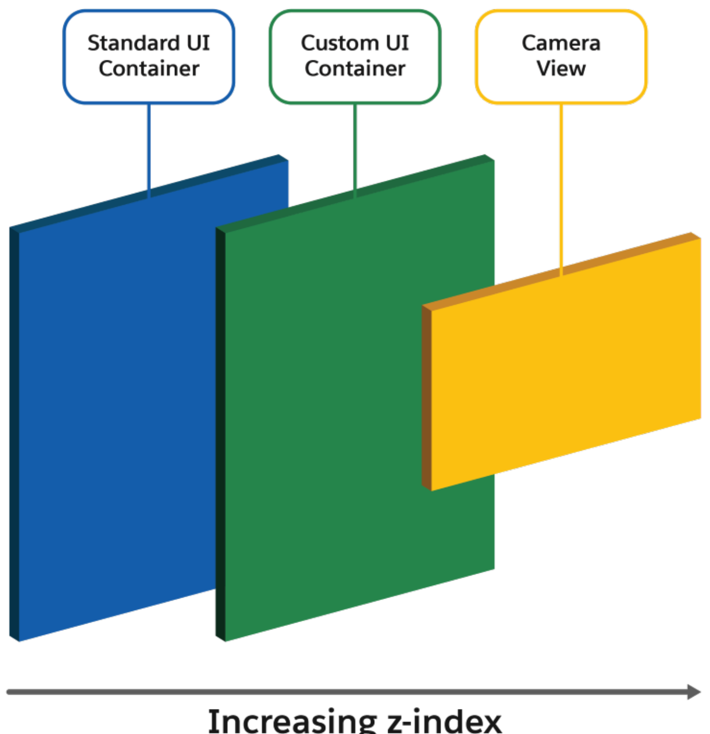
You can use a static HTML string, or generate the HTML at runtime. There are only a few requirements.

- The `<header>` element is required, and should not be modified. Confine your customizations to the `<html>` and child elements of the page.
- You should also provide a UI element to dismiss or cancel the scanning session. See Dismiss the Scanner.

To apply your custom UI to BarcodeScanner, set the `backgroundViewHTML` property of the `BarcodeScannerOptions` configuration object to the string value of your HTML page, including the `<header>`. Then provide `BarcodeScannerOptions` when calling `beginCapture()`.

UI Customization Layers

Your custom UI overlays — and completely hides — all parts of the standard BarcodeScanner UI. The following diagram illustrates the layers of the scanner UI as rendered by BarcodeScanner:



Dismiss the Scanner

Important: We recommend using the modern `scan()` and `dismiss()` API functions in your LWC scanning code to streamline your development experience. The legacy API functions `beginCapture()`, `resumeCapture()`, and `endCapture()` are still available, but will be retired in a future release. See Understand BarcodeScanner Modern and Legacy APIs on page 35 for additional details.

When you define a custom UI, you replace the standard scanner dismissal control. In your custom UI, it's *your* responsibility to handle dismissing the scanner. You can dismiss the scanner two different ways:

- *Programatically*: by calling `endCapture` to dismiss the scanner UI.
- *UI Triggered*: by adding an element to your custom UI that, when triggered, navigates to a special URL: `nimbusbarcodescanner://dismiss`.

  For example, here's a simple text link that closes the scanner when tapped

```
<a href="nimbusbarcodescanner://dismiss">Dismiss</a>
```

Place this anywhere in your custom user interface that makes sense.

Considerations

- The camera view is always placed in the center of the device screen (horizontally and vertically), and is superimposed onto your custom UI. Consider this when designing your custom UI, and avoid having essential parts of the UI obscured by the camera view.
- The custom UI is rendered in a separate webview container than the main webview container that hosts your Lightning web component. The HTML that renders your custom UI can't reference or access elements or objects that are defined in your component.
- Note: This plugin is not supported in the Field Service mobile app.

Compatibility and Requirements

BarcodeScanner is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when BarcodeScanner runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a web browser, whether the browser runs on a desktop or mobile device.**

BarcodeScanner is available in Lightning apps distributed using:

- Salesforce Mobile app
- Mobile Publisher for Salesforce App
- Mobile Publisher for Experience Cloud

Note: The Field Service Mobile app provides an alternative implementation of BarcodeScanner. See Scan Barcodes on a Mobile Device in the *Field Service Developer Guide* for details.

BarcodeScanner is fully functional when used in a Lightning app or Lightning site run from one of these Salesforce apps on a compatible iOS or Android mobile device. See Requirements for the Salesforce Mobile App, or the requirements page for your target mobile app, for specific device and operating system requirements.

BarcodeScanner is **not** fully functional when running on other devices, such as a desktop, or when running in a web browser, even on a mobile device. It **requires** one of the mobile apps listed above. The BarcodeScanner API is accessible in Lightning Experience on all devices, so your code won't fail due to missing functions. However, when your app runs in a browser—desktop or mobile—it can use only BarcodeScanner constants and utility functions. Attempting any scanning operation will fail.

Supported Barcode Types

BarcodeScanner can recognize the following standard barcode symbologies. (References not affiliated with Salesforce.)

Barcode Symbology Standard	BarcodeScanner Type (`barcodeTypes`)
Code 128	`CODE_128`
Code 39	`CODE_39`
Code 93	`CODE_93`
Data Matrix	`DATA_MATRIX`
EAN-13 / GTIN-13	`EAN_13`
EAN-8 / GTIN-8	`EAN_8`

Barcode Symbology Standard	BarcodeScanner Type (`barcodeTypes`)
Interleaved 2 of 5	`ITF`
PDF417	`PDF_417`
QR-Code	`QR`
UPC-A / GTIN-12	`UPC_A`
UPC-E / GTIN-12	`UPC_E`

To access or compare barcode types in code, use the `barcodeTypes` constant.

BarcodeScanner doesn't attempt to interpret the value found in a barcode. The contents of the barcode are decoded into a string value. It's up to the controlling component or application to further parse and interpret the result and decide what to do with it. For more information about barcode standards and symbologies, see Barcoding for Beginners (not affiliated with Salesforce).

Considerations and Limitations

Keep the following in mind when developing features that use the BarcodeScanner API.

Device Limitations

- BarcodeScanner requires the use of the mobile device camera. The user must grant your app access to the camera. The exact user experience is governed by the platform. The request happens automatically on first use, and is managed by the device itself, but you should plan for it when designing the user experience of your app.
- In Android 11 or later, if the user taps "Deny" for permission to access the Contacts app more than once during the app's lifetime of installation on a device, the user won't see the system permissions dialog again. Tapping Deny multiple times implicitly chooses the "don't ask again" option.

  In previous versions of Android, users would see the system permissions dialog each time the app requested permission unless the user had previously selected "don't ask again". This change in Android 11 discourages repeated requests for permissions that users have chosen to deny.

  If the user has denied permission to access the Contacts app and needs to change their permissions to allow access, they can do so in their device's settings.
- BarcodeScanner doesn't implement scanning itself. Instead, it makes available the scanning feature of the underlying platform (Android or iOS). While the features provided by BarcodeScanner are the same across both platforms, it *is* subject to some platform-specific quirks and minor differences.
  - If you can't get a clear picture of the barcode, it can't be recognized. The quality of the device camera affects barcode recognition. A damaged or low-quality camera lens or focusing system, poor lighting, motion, and other factors can make it difficult or impossible to get a clear picture of a barcode.
  - The quality of the barcode affects barcode recognition. Specifically, damaged or obscured barcodes are hard to recognize successfully.
- If you're having trouble getting BarcodeScanner to recognize a barcode, try the following:
  - First, verify that the barcode type is one of the supported barcode symbologies. There are other barcode types that aren't supported.
  - Second, verify that you've configured BarcodeScanner to recognize the expected symbology. See `BarcodeScannerOptions` in BarcodeScanner Data Types for configuration details.

- Finally, check whether another app on the same device is able to recognize the barcode. If the standard camera app on the device can't recognize the barcode, neither can BarcodeScanner.

Development Considerations

- BarcodeScanner requires access to camera hardware. To test scanning during development, use actual, physical devices.
  - The Android emulator can simulate camera hardware by using a webcam on your development system. To do so, edit the camera configuration for your Android Virtual Device, in the advanced settings panel. However, the camera built into most laptops is much lower quality than what's found on modern mobile phones. A low-quality camera limits the usefulness of testing barcode recognition.
  - The iOS simulator doesn't provide access to simulated camera hardware at all.

You can certainly develop the user experience for your component on a desktop or laptop development system. But be sure to test scanning functionality on the physical devices on which you plan to deploy your Lightning app.

BarcodeScanner Considerations

Be aware of the following considerations when using BarcodeScanner in your Lightning app.

- BarcodeScanner is built on top of mobile operating system features. BarcodeScanner's scanning capabilities therefore depend on Android or iOS features, which are subject to change beyond our control. When mobile operating system features change, the behavior of BarcodeScanner can change without notice.
- BarcodeScanner provides haptic feedback (a short vibration) after a successful scan on iOS devices. There's no haptic feedback on Android devices.

Access a Mobile Device's Biometrics Capabilities

A Lightning web component can use a device's biometrics functionality to prompt a user to confirm their identity. When these biometrics-related actions occur, the result is returned to the Lightning web component that invoked it.

Biometrics checks are managed locally on the mobile device, and don't need a network connection. However, BiometricsService requires access to platform-specific APIs that are available only within compatible Salesforce mobile apps.

Important: BiometricsService does not and cannot function when running in a web browser, whether on a desktop or mobile device.

IN THIS SECTION:

BiometricsService User Experience

Your component can deliver any user experience you desire, but there's a common flow for any component that calls for a biometrics check.

Use the BiometricsService API

To develop a Lightning web component with biometrics-checking features, use the BiometricsService API as your method for accessing a device's native biometrics functionality.

BiometricsService Example

Here's a basic example of a Lightning web component that uses a device's biometrics capabilities to verify device ownership.

Compatibility and Requirements

BiometricsService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when BiometricsService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

Considerations and Limitations

Consider these guidelines and limitations when developing features that use the BiometricsService API.

SEE ALSO:

Lightning Web Components Developer Guide: BiometricsService API

BiometricsService User Experience

Your component can deliver any user experience you desire, but there's a common flow for any component that calls for a biometrics check.

- Your user performs an action that triggers a biometrics check.
- The OS prompts the user to confirm their identity via a biometrics check.
- The OS provides a success message when the biometrics are confirmed, and continues the operation that the user initiated before the biometrics check. The OS provides an error message if the biometrics check isn't successful.

In some of these examples, BiometricsService is only a part of the complete solution. Determining where in your app experience a biometrics check is appropriate, implementing it as part of a user flow, and so on, are other parts that you must implement yourself.

Use the BiometricsService API

To develop a Lightning web component with biometrics-checking features, use the BiometricsService API as your method for accessing a device's native biometrics functionality.

1. Import BiometricsService into your component definition to make the BiometricsService API functions available to your code.
2. Test to make sure BiometricsService is available before you call for a biometrics check.
3. Use the feature functions to prompt app users for biometrics checks.

Add BiometricsService to a Lightning Web Component

In your component's JavaScript file, import BiometricsService using the standard JavaScript `import` statement. Specifically, import the `getBiometricsService()` factory function from the `lightning/mobileCapabilities` module, like so:

```
import { getBiometricsService } from 'lightning/mobileCapabilities';
```

After it's imported into your component, use the factory function to get an instance of BiometricsService. With your BiometricsService instance, use the utility functions and constants to verify availability. Then use the feature functions to perform the associated functionality.

Test BiometricsService Availability

BiometricsService depends on physical device hardware and platform features. A component that uses BiometricsService renders without errors on a desktop computer or in a mobile browser, but biometrics-checking functions fail. To avoid these errors, test if BiometricsService functionality is available before you use it.

```
handleCheckBiometricsClick(event) {
    const myBiometricsService = getBiometricsService();
    if(myBiometricsService.isAvailable()) {
        // Perform biometrics-checking operations
    }
    else {
```

```
        // BiometricsService not available, or consuming app hasn’t implemented it

        // Not running on hardware with biometrics functionality, etc.
        // Handle with message, error, beep, and so on
    }
}
```

Check Biometrics Availability and Configuration

It's simple to confirm a device's biometrics functionality in your Lightning web component using BiometricsService. First, use `isBiometricsReady()` to confirm that a device has biometrics functionality and that it's set up for use. Then, process the result in whatever manner you wish.

For example:

```
// Check for device biometrics functionality, console log the results
myBiometricsService.isBiometricsReady(options)
.then((results) => {
    console.log(results);
})
.catch((error) => {
    // Handle cancellation or other errors here
    console.error('Error code: ' + error.code); +
    console.error('Error message: ' + error.message);
});
```

Prompt a Biometric Check

Prompt a device biometrics check with the `checkUserIsDeviceOwner()` function. First, call the `checkUserIsDeviceOwner()` function, optionally including a `BiometricsServiceOptions` parameter. Then, handle the outcome in whatever manner you wish.

For example:

```
// Get events from a specified date range from the specified calendar(s), and then  process
 them
myBiometricsService.checkUserIsDeviceOwner(options)
.then((results) => {
    // Do something with the event(s) data
    this.events = results;
    console.log(results);
})
.catch((error) => {
    // Handle cancellation or other errors here
    this.events = [];
    console.error('Error code: ' + error.code); +
    console.error('Error message: ' + error.message);
});
```

SEE ALSO:

Lightning Web Components Developer Guide: BiometricsService API

BiometricsService Example

BiometricsService Example

Here's a basic example of a Lightning web component that uses a device's biometrics capabilities to verify device ownership.

The component's HTML template is minimal, with a "Verify" button to initiate the biometrics check.

```
<template>
  <lightning-card title="Biometrics Service Demo" icon-name="custom:privately_shared">
    <div class="slds-var-m-around_medium">
      Use device biometrics capabilities to verify current user is indeed device owner:
      <lightning-button
        variant="brand"
        label="Verify"
        title="Verify device ownership using biometrics"
        onclick={handleVerifyClick}
        class="slds-var-m-left_x-small">
      </lightning-button>
    </div>
    <div class="slds-var-m-around_medium">
      <lightning-formatted-text value={status}></lightning-formatted-text>
    </div>
  </lightning-card>
</template>
```

This example simply uses BiometricsService to prompt the user to complete a biometrics check. A status message is returned, indicating whether the check was successful or not.

```
import { LightningElement } from 'lwc';
import { getBiometricsService } from 'lightning/mobileCapabilities';

export default class NimbusPluginBiometricsService extends LightningElement {
    status;
    biometricsService;

    connectedCallback() {
      this.biometricsService = getBiometricsService();
    }

    handleVerifyClick() {
      if (this.biometricsService.isAvailable()) {
        const options = {
          permissionRequestBody: "Required to confirm device ownership.",
          additionalSupportedPolicies: ['PIN_CODE']
        };
        this.biometricsService.checkUserIsDeviceOwner(options)
          .then((result) => {
            // Do something with the result
            if (result === true) {
              this.status = "□ Current user is device owner."
            } else {
              this.status = "□□ Current user is NOT device owner."
            }
          })
          .catch((error) => {
            // Handle errors
          this.status = 'Error code: ' + error.code + '\nError message: ' + error.message;
```

```
          });
        } else {
          // service not available
          this.status = 'Problem initiating Biometrics service. Are you using a mobile
device?';
        }
      }
}
```

SEE ALSO:

Use the BiometricsService API

Compatibility and Requirements

BiometricsService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when BiometricsService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

BiometricsService is available in Lightning apps distributed using:

- Salesforce Mobile app
- Mobile Publisher for Experience Cloud

BiometricsService is fully functional when used in a Lightning app or Lightning site that's run from a compatible Salesforce mobile app on a compatible iOS or Android mobile device. See Requirements for Mobile Publisher for Experience Cloud, or the requirements page for your target mobile app for specific device and operating system requirements.

BiometricsServiceis **not** fully available when running on other devices, such as a desktop, or when running in a web browser, even on a mobile device. It **requires** one of the apps listed above. The BiometricsService API is accessible in Lightning Experience on all devices, so your code won't fail due to missing functions. However, when your app runs in a browser—desktop or mobile—it can use only BiometricsService constants and utility functions. Attempting any biometrics related operation will fail.

Considerations and Limitations

Consider these guidelines and limitations when developing features that use the BiometricsService API.

Device Limitations

BiometricsService doesn't manage biometrics data and checking itself. Instead, it makes available certain biometrics capabilities of the underlying platform (Android or iOS) and hardware (phone or other mobile device). While the features provided by BiometricsService are the same across both platforms, they're subject to some platform-specific quirks and minor differences.

Development Considerations

You can certainly develop the user experience for your component on a desktop or laptop development system. But be sure to test biometrics functionality on the physical devices on which you plan to deploy your Lightning app.

BiometricsService Considerations

BiometricsService is built on top of mobile operating system and device features. BiometricsService's capabilities therefore depend on Android or iOS features, which are subject to change beyond our control. When mobile operating system features change, the behavior of BiometricsService can change without notice.

Manage Calendar Events on a Mobile Device

A Lightning web component can use a device's calendar functionality to create, read, update, and delete calendar events to and from the device. When these calendar-related actions occur, the event data is returned to the Lightning web component that invoked it.

Calendar events are managed locally on the mobile device, and don't need a network connection. However, CalendarService requires access to platform-specific APIs that are available only within compatible Salesforce mobile apps.

Important: CalendarService does not and cannot function when running in a web browser, whether on a desktop or mobile device.

IN THIS SECTION:

CalendarService User Experience

Your component can deliver any user experience you like. There are a number of common calendar-based features where CalendarService might be suitable.

Use the CalendarService API

To develop a Lightning web component with calendar-based features, use the CalendarService API as your method for accessing a device's native calendar functionality.

CalendarService Example

Here's a basic example of a Lightning web component that displays calendar events and allows the user to perform basic calendar-related functions.

Compatibility and Requirements

CalendarService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It requires access to device hardware and device platform APIs. This access is only available when CalendarService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

Considerations and Limitations

Keep the following in mind when developing features that use the CalendarService API.

SEE ALSO:

Lightning Web Components Developer Guide: CalendarService API

CalendarService User Experience

Your component can deliver any user experience you like. There are a number of common calendar-based features where CalendarService might be suitable.

Here are a few examples of common calendar-based features:

- View all calendar events for a specified date or date range
- Add a new calendar event to a device's calendar
- Check if a newly scheduled event conflicts with any events on a device's calendar
- Schedule a reminder to avoid missing important events
- Compare an event on a device calendar to another calendar
- Perform an action when a calendar event begins
- Read, change, or remove an existing calendar event

In some of these examples, CalendarService is only a part of the complete solution. Displaying event data, checking a calendar for availability, and so on are other parts that you need to implement yourself.

Use the CalendarService API

To develop a Lightning web component with calendar-based features, use the CalendarService API as your method for accessing a device's native calendar functionality.

1. Import CalendarService into your component definition to make the CalendarService API functions available to your code.
2. Test to make sure CalendarService is available before you call calendar-related functions.
3. Use the calendar functions to create, read, update, and delete calendar events.

Add CalendarService to a Lightning Web Component

In your component's JavaScript file, import CalendarService using the standard JavaScript `import` statement. Specifically, import the `getCalendarService()` factory function from the `lightning/mobileCapabilities` module, like so:

```
import { getCalendarService } from 'lightning/mobileCapabilities';
```

After it's imported into your component, use the factory function to get an instance of CalendarService. With your CalendarService instance, use the utility functions and constants to verify availability. Then use calendar-related functions to perform the associated functionality.

Test CalendarService Availability

CalendarService depends on physical device hardware and platform features. A component that uses CalendarService renders without errors on a desktop computer or in a mobile browser, but calendar-related functions fail. To avoid these errors, test if CalendarService functionality is available before you use it.

```
handleManageCalendarEventsClick(event) {
   const myCalendarService = getCalendarService();
   if(myCalendarService.isAvailable()) {
     // Perform calendar-related operations
   }
   else {
     // CalendarService not available, or consuming app hasn't implemented it
     // Not running on hardware with a native calendar application, etc.
     // Handle with message, error, beep, and so on
   }
 }
```

Get Calendars

It's simple to import device calendars into your Lightning web component using CalendarService. First, use `getCalendars()` to enable access within your component to available device calendars. Then, process the calendar data in whatever manner you wish.

For example:

```
// Get access to device calendars and process them
myCalendarService.getCalendars(options)
  .then((results) => {
```

```
    // do something with the calendar(s) data
    this.calendars = results;
    console.log(results);
  })
  .catch((error) => {
    // Handle cancellation or other errors here
    this.calendars = [];
    console.error('Error code: ' + error.code); +
    console.error('Error message: ' + error.message);
  });
```

Get Events

Fetch all calendar events within a specified date range with the `getEvents()` function. First, call the `getEvents()` function with the necessary parameters. Then, process or display event data in whatever manner you wish.

For example, if your calendar component has a "week view" and a "month view," the `startDateSecondsUTC` and `endDateSecondsUTC` parameters need to be adjusted to capture the appropriate date range to be displayed. If you don't want to display events from all mobile device calendars, specify the ones you want to work with in the `calendars[]` array.

For example:

```
// Get events from a specified date range on the specified calendar(s), and process them
myCalendarService.getEvents(startDateSecondsUTC, endDateSecondsUTC, calendars, options)
  .then((results) => {
    // Do something with the event(s) data
    this.events = results;
    console.log(results);
  })
  .catch((error) => {
    // Handle errors here
    this.events = [];
    console.error('Error code: ' + error.code); +
    console.error('Error message: ' + error.message);
  });
```

Create a Calendar Event

Create and add new calendar events to a mobile device with the `addEvent()` function. First, call the `addEvent()` function with the necessary parameters. Then, handle a successful outcome with a success message, or in any manner you wish. For more information on what the `addEvent()` function returns, see Implicit Data Coercion on page 67.

For example:

```
// Add an event to a mobile device calendar, and then handle a success
myCalendarService.addEvent(event, options)
  .then((results) => {
    // Do something with the event(s) data
    this.newEvent = results;
    console.log(results);
  })
  .catch((error) => {
```

```
    // Handle cancellation or other errors here
    console.error('Error code: ' + error.code); +
    console.error('Error message: ' + error.message);
  });
```

Update a Calendar Event

Update calendar events on a mobile device with the `updateEvent()` function. First, call the `updateEvent()` function with the necessary parameters. Then, handle a successful outcome with a success message, or in any manner you wish. For more information on what the `updateEvent()` function returns, see Implicit Data Coercion on page 67.

For example:

```
// Update an event on a mobile device calendar, and then handle a success
myCalendarService.updateEvent(event, options)
   .then((results) => {
     // Do something with the event(s) data
     this.updatedEvent = results;
     console.log(results);
   })
   .catch((error) => {
     // Handle cancellation or other errors here
     console.error('Error code: ' + error.code); +
     console.error('Error message: ' + error.message);
   });
```

Warning: Using `updateEvent()` is an inherently dangerous action, as it allows event data to be irreversibly altered. Use caution when using this functionality in your component. At an absolute minimum, consider adding a confirmation window for your users that clearly states the outcome of the action, with an option for them to cancel if they wish.

Remove a Calendar Event

Remove calendar events on a mobile device with the `removeEvent()` function. First, call the `removeEvent()` function with the necessary parameters. Then, handle a successful outcome with a success message, or in any manner you wish.

For example:

```
// Remove an event on a mobile device calendar, and then handle a success
myCalendarService.removeEvent(event, options)
  .then((results) => {
    // Handle successful deletion here
    console.log('Event successfully deleted!');
  })
  .catch((error) => {
    // Handle cancellation or other errors here
    console.error('Error code: ' + error.code); +
    console.error('Error message: ' + error.message);
  });
```

Warning: Using `removeEvent()` is an inherently dangerous action, as it allows event data to be irreversibly altered. Use caution when using this functionality in your component. At an absolute minimum, consider adding a confirmation window for your users that clearly states the outcome of the action, with an option for them to cancel if they wish.

Implicit Data Coercion

When passing in event data to a CalendarService function (namely the `addEvent()` and `updateEvent()` functions), CalendarService can change some event data before returning it. This behavior, referred to here as implicit data coercion, occurs when CalendarService adjusts the value of a property as a result of a user's change of another property.

For example, if a new all-day event is added (or if an existing event is updated to be all-day) and a start time or end time is specified, CalendarService rejects the times. Instead, the start time is set to 00:00:00 (12:00:00 AM) and the end time to 23:59:59 (11:59:59 PM).

Implicit data coercion can also occur when changing details of recurring events. Any time recurring events are changed, their old IDs are overwritten and replaced with newly generated IDs.

Keep this in mind when using the `addEvent()` and `updateEvent()` functions. Your code should always use the coerced event data returned by CalendarService. Using raw (uncoerced) event data in your component can lead to errors and incorrect behavior.

SEE ALSO:

Lightning Web Components Developer Guide: CalendarService API

CalendarService Example

CalendarService Example

Here's a basic example of a Lightning web component that displays calendar events and allows the user to perform basic calendar-related functions.

The component's HTML template is minimal, with a "main" display view that lists calendar events and a "detail" display view that shows an event's details.

```
<template>
   <!-- Main View -->
   <template if:false={detailViewIsOpen}>
       <lightning-card title="Today's Events" icon-name="utility:dayview">
           <template for:each={todayEvents} for:item="item">
               <div class="slds-var-p-horizontal_medium slds-var-p-vertical_x-small"
key={item.id} onclick={showDetailView} data-id={item.id}>
                   <p class="slds-text-heading_small"><b>{item.title}</b></p>
                   <p class="slds-text-heading_small">{item.startTimeDisplay} —
{item.endTimeDisplay}</p>
               </div>
           </template>
       </lightning-card>
   </template>

       <!-- Detail View -->
   <template if:true={detailViewIsOpen}>
      <div class="slds-var-p-around_medium" style="background-color: white; border-radius:
 4px;">
           <table>
               <tbody>
                   <tr>
                       <td class="slds-align-top" width="1">
                           <lightning-icon icon-name="utility:chevronleft"
onclick={hideDetailView}></lightning-icon>
                       </td>
```

```
                        <td class="sldx-align-top slds-text-heading_medium
slds-var-p-bottom_medium slds-align_absolute-center">Event Details</td>
                        <td class="slds-align-top" width="1">
                            <lightning-button-menu alternative-text="Show menu"
variant="border-filled" menu-alignment="auto">
                                <lightning-menu-item prefix-icon-name="utility:add"
value="Add" label="Add to Device Calendar" onclick={addCalendarEvent}></lightning-menu-item>

                                <lightning-menu-item prefix-icon-name="utility:edit"
value="Update" label="Update in Device Calendar"
onclick={updateCalendarEvent}></lightning-menu-item>
                                <lightning-menu-item prefix-icon-name="utility:delete"
value="Delete" label="Remove from Device Calendar"
onclick={deleteCalendarEvent}></lightning-menu-item>
                            </lightning-button-menu>
                        </td>
                    </tr>
                    <tr>
                        <td colspan="3">
                            <ul class="slds-has-dividers_bottom-space">
                                <li class="slds-item">
                                    <span
class="slds-text-heading_small"><b>{selectedItem.title}</b></span><br>
                                    <span
class="slds-text-heading_small">{selectedItem.startTimeDisplay} —
{selectedItem.endTimeDisplay}</span>
                                </li>
                                <li class="slds-item">
                                    <span
class="slds-text-heading_small">Reminders</span><br>
                                        <template for:each={selectedItem.alarmsDisplay}
for:item="alarm">
                                            <span class="slds-text-body_regular"
key={alarm}>{alarm}<br></span>
                                        </template>
                                </li>
                                <li class="slds-item">
                                    <span
class="slds-text-heading_small">Location</span><br>
                                    <span
class="slds-text-body_regular">{selectedItem.location}</span>
                                </li>
                                <li class="slds-item">
                                    <span
class="slds-text-heading_small">Attendees</span><br>
                                        <template for:each={selectedItem.attendees}
for:item="attendee">
                                            <span class="slds-text-body_regular"
key={attendee.name}>{attendee.name} ({attendee.email})<br></span>
                                        </template>
                                </li>
                                <li class="slds-item">
                                  <span class="slds-text-heading_small">Notes</span><br>
```

```
                                        <span
class="slds-text-body_regular">{selectedItem.notes}</span>
                                    </li>
                                </ul>
                            </td>
                        </tr>
                    </tbody>
                </table>
                </div>
            </template>
        </template>
```

This example simply uses CalendarService to display events, and allows you to perform simple actions on calendar items. A status message is returned when there's an error. In this example, the events are hard-coded, rather than fetched via API calls from a Salesforce org. You'll need to build functionality to fetch event data from your Salesforce org as part of your component.

```
import { api, LightningElement } from 'lwc';
import { getCalendarService } from 'lightning/mobileCapabilities';
import LightningAlert from 'lightning/alert'
import LightningConfirm from 'lightning/confirm';

export default class CalendarForToday extends LightningElement {

    todayEvents = [];
    detailViewIsOpen = false;
    selectedItem = null;
    selectedItemIndex = -1;
    calendarPermissionRationaleText = "Allow access to your calendar to enable calendar
event processing.";
    calendarService;

    connectedCallback() {
        console.log("Start connected callback");

        try {
            this.calendarService = getCalendarService();
            this.todayEvents = this.getTodayEvents();
            this.todayEvents.forEach(item => this.generateDisplayFields(item));
           console.log(`End connected callback with ${this.todayEvents.length} events for
 today.`);
        } catch (err) {
            console.log(`connectedCallback failed with error: ${err}`);
        }
    }

    showDetailView(event) {
        const id = event.currentTarget.dataset.id;
        this.selectedItemIndex = this.todayEvents.findIndex(item => item.id === id);
        if (this.selectedItemIndex != -1) {
            this.selectedItem = this.todayEvents[this.selectedItemIndex];
        } else {
            this.selectedItem = null;
        }
```

```
        this.detailViewIsOpen = this.selectedItem != null;
    }

    hideDetailView() {
        this.detailViewIsOpen = false;
        this.selectedItem = null;
        this.selectedItemIndex = -1;
    }

    addCalendarEvent() {
        if (this.calendarService.isAvailable() && this.selectedItemIndex != -1 &&
this.selectedItem) {
            const options = {
                "permissionRationaleText" : this.calendarPermissionRationaleText
            };

            console.log(`options: ${JSON.stringify(options)}`);
            console.log(`Adding selectedItem: ${JSON.stringify(this.selectedItem)}`);

            this.calendarService.addEvent(this.selectedItem, options)
            .then((sanitizedEvent) => {
                this.generateDisplayFields(sanitizedEvent);
                this.selectedItem = sanitizedEvent;
                this.todayEvents[this.selectedItemIndex] = sanitizedEvent;
                this.showSuccessAlert("Add Event", "Event was added successfully to the
device default calendar.");
                console.log(`sanitizedEvent: ${JSON.stringify(sanitizedEvent)}`);
            })
            .catch((error) => {
                console.error(error);
                this.showFailureAlert("Add Event", `There was a problem adding the event
to the device default calendar: ${error.message}`);
            });
        } else {
            console.log("Calendar Service Is Not Available");
            this.showFailureAlert("Add Event", "Calendar Service is not available.");
        }
    }

    updateCalendarEvent() {
        if (this.calendarService.isAvailable() && this.selectedItemIndex != -1 &&
this.selectedItem) {

            // For this sample code, we've hard-coded some trivial changes

            this.selectedItem.title += " - Updated";
            this.selectedItem.notes += " - Updated";

            const options = {
                "permissionRationaleText" : this.calendarPermissionRationaleText,
                "span" : "ThisEvent"
            };

            console.log(`options: ${JSON.stringify(options)}`);
```

```
            console.log(`Updating selectedItem: ${JSON.stringify(this.selectedItem)}`);

            this.calendarService.updateEvent(this.selectedItem, options)
            .then((sanitizedEvent) => {
                this.generateDisplayFields(sanitizedEvent);
                this.selectedItem = sanitizedEvent;
                this.todayEvents[this.selectedItemIndex] = sanitizedEvent;
                this.showSuccessAlert("Update Event", "Event was updated successfully in
the device default calendar.");
                console.log(`sanitizedEvent: ${JSON.stringify(sanitizedEvent)}`);
            })
            .catch((error) => {
                console.error(error);
                this.showFailureAlert("Update Event", `There was a problem updating the
event in the device default calendar: ${error.message}`);
            });
        } else {
            console.log("Calendar Service Is Not Available");
            this.showFailureAlert("Update Event", "Calendar Service is not available.");
        }
    }

    deleteCalendarEvent() {
        if (this.calendarService.isAvailable() && this.selectedItemIndex != -1 &&
this.selectedItem) {
            LightningConfirm.open(
                {
                    label: "Delete Event",
                    message: "Are you sure you want to delete this event?",
                    theme: "warning"
                }
            ).then((response) => {
                if (response === true) {
                    const options = {
                        "permissionRationaleText" : this.calendarPermissionRationaleText,

                        "span" : "ThisEvent"
                    };

                    console.log(`options: ${JSON.stringify(options)}`);
                    console.log(`Deleting selectedItem:
${JSON.stringify(this.selectedItem)}`);

                    this.calendarService.removeEvent(this.selectedItem, options)
                    .then(() => {
                        this.todayEvents.splice(this.selectedItemIndex, 1);
                        this.hideDetailView();
                     this.showSuccessAlert("Delete Event", "Event was removed successfully
 from the device default calendar.");
                    })
                    .catch((error) => {
                        console.error(error);
                      this.showFailureAlert("Delete Event", `There was a problem removing
 the event from the device default calendar: ${error.message}`);
```

```
                    });
                }
            });
        } else {
            console.log("Calendar Service Is Not Available");
            this.showFailureAlert("Delete Event", "Calendar Service is not available.");
        }
    }

    getTodayEvents() {
        // For this sample code, we've hard-coded some made-up values.
        // Your component should fetch events from your SF org and convert them to the
following object format.
        const events = [
            {
                id: "event_id_1", // will be overwritten after a call to
calendarService.addEvent()
                isAllDay: false,
                startDateSecondsUTC: this.getTodayTimestampAtTimeOfDay(8, 0, 0), // 8 AM
                endDateSecondsUTC: this.getTodayTimestampAtTimeOfDay(10, 0, 0), // 10 AM
                availability: "Busy",
                status: "Confirmed",
                calendarId: null, // will be assigned to the default device calendar
                title: "Team Meeting",
                location: "3514 Ruckman Road, San Francisco, CA 94105",
                notes: "Discussing customer request for new calendar feature",
                alarms: [{relativeOffsetSeconds: 600}], // 10 mins before event
                attendees: [
                    { name: "Jamal Booker", email: "jbooker_fake_email@email.com", role:
"Required", status: "Accepted" },
                    { name: "Robert Bullard", email: "bob.bullard.fake.email@email.com",
role: "Required", status: "Pending" },
                    { name: "Gordon Chu", email: "gordonchu736251_email@email.com", role:
 "Optional", status: "Declined" },
                ],
                recurrenceRules: null
            },
            {
                id: "event_id_2", // will be overwritten after a call to
calendarService.addEvent()
                isAllDay: false,
              startDateSecondsUTC: this.getTodayTimestampAtTimeOfDay(10, 30, 0), // 10:30
 AM
                endDateSecondsUTC: this.getTodayTimestampAtTimeOfDay(11, 0, 0), // 11 AM
                availability: "Busy",
                status: "Confirmed",
                calendarId: null, // will be assigned to the default device calendar
                title: "Quarterly Review",
                location: "2135 Alpha Avenue, Fernandina Beach, FL 32034",
                notes: "Reviewing results of Q2 and planning Q3",
                alarms: [{relativeOffsetSeconds: 1800}], // 30 mins before event
                attendees: [
                   { name: "Alex Driskel", email: "adriskell_fake_email@email.com", role:
 "Required", status: "Accepted" },
```

```
                    { name: "Kim Friedman", email:
"nothing_is_kimpossible_fake_email@email.com", role: "Required", status: "Tentative" },
                    { name: "April Guthman", email: "guthman.april.fake@email.com", role:
 "Required", status: "Pending" },
                    { name: "Leif Hansen", email: "leifhansenfakeemail@email.com", role:
"Required", status: "Accepted" },
                ],
                recurrenceRules: null
            },
            {
                id: "event_id_3", // will be overwritten after a call to
calendarService.addEvent()
                isAllDay: false,
                startDateSecondsUTC: this.getTodayTimestampAtTimeOfDay(11, 0, 0), // 11
AM
                endDateSecondsUTC: this.getTodayTimestampAtTimeOfDay(12, 0, 0), // 12 PM
                availability: "Busy",
                status: "Confirmed",
                calendarId: null, // will be assigned to the default device calendar
                title: "Portfolio Checklist",
                location: "2281 Radford Street, Louisville, KY 40291",
                notes: "Creating a guide to help sales in compiling a strong portfolio",
                alarms: [{relativeOffsetSeconds: 600}], // 10 mins before event
                attendees: [
                  { name: "Marie Hill", email: "marieriefake@email.com", role: "Required",
 status: "Accepted" },
                    { name: "Foua Khang", email: "khanghangemail@email.com", role:
"Required", status: "Accepted" },
                    { name: "Mindy Lee", email: "mindyfakeemaillee@email.com", role:
"Required", status: "Accepted" },
                ],
                recurrenceRules: null
            },
            {
                id: "event_id_4", // will be overwritten after a call to
calendarService.addEvent()
                isAllDay: false,
               startDateSecondsUTC: this.getTodayTimestampAtTimeOfDay(12, 30, 0), // 12:30
 PM
                endDateSecondsUTC: this.getTodayTimestampAtTimeOfDay(13, 30, 0), // 1:30
PM
                availability: "Tentative",
                status: "Tentative",
                calendarId: null, // will be assigned to the default device calendar
                title: "Lunch with Jennifer West",
                location: "171 2nd St, San Francisco, CA 94105",
                notes: "Bring latest version of contract",
                alarms: [{relativeOffsetSeconds: 1800}], // 30 mins before event
                attendees: [
                    { name: "Awanasa Locklear", email: "this_is_awanasa_fake@email.com",
role: "Required", status: "Tentative" },
                    { name: "Elena Nieto", email: "elenasfakeemail@email.com", role:
"Required", status: "Tentative" },
                ],
```

```
                recurrenceRules: null
            },
            {
                id: "event_id_5", // will be overwritten after a call to
calendarService.addEvent()
                isAllDay: false,
               startDateSecondsUTC: this.getTodayTimestampAtTimeOfDay(14, 30, 0), // 2:30
 PM
                endDateSecondsUTC: this.getTodayTimestampAtTimeOfDay(15, 30, 0), // 3:30
PM
                availability: "Tentative",
                status: "Tentative",
                calendarId: null, // will be assigned to the default device calendar
                title: "Sales Workgroup",
                location: "3270 Armbrester Drive, Gardena, CA 90248",
                notes: "Discuss the new customer opportunities",
                alarms: [{relativeOffsetSeconds: 1800}], // 30 mins before event
                attendees: [
                    { name: "Raul Nieto", email: "raul.fake.nieto@email.com", role:
"Required", status: "Accepted" },
                   { name: "Salome Ofodu", email: "salomesalomefakeemail@email.com", role:
 "Required", status: "Tentative" },
                    { name: "Justus Pardo", email: "justuspardofake@email.com", role:
"Required", status: "Tentative" },
                    { name: "Gorav Patel", email: "gpatel.fake.email@email.com", role:
"Required", status: "Pending" },
                ],
                recurrenceRules: null
            },
            {
                id: "event_id_6", // will be overwritten after a call to
calendarService.addEvent()
                isAllDay: false,
               startDateSecondsUTC: this.getTodayTimestampAtTimeOfDay(15, 30, 0), // 3:30
 PM
                endDateSecondsUTC: this.getTodayTimestampAtTimeOfDay(17, 30, 0), // 5:30
PM
                availability: "Busy",
                status: "Confirmed",
                calendarId: null, // will be assigned to the default device calendar
                title: "Executive Team",
                location: "56 Main Street, Seattle, WA 98119",
                notes: "Report on Q2 sales and new leads",
                alarms: [{relativeOffsetSeconds: 600}, {relativeOffsetSeconds: 3600}], //
 10 mins and 1 hour before event
                attendees: [
                   { name: "Florentina Perez", email: "florperfakeemail@email.com", role:
 "Required", status: "Accepted" },
                   { name: "Harryette Randall", email: "doubleletteremailfake@email.com",
 role: "Required", status: "Accepted" },
                    { name: "Sofia Rivera", email: "fakeemailforsofie@email.com", role:
"Required", status: "Accepted" },
                ],
                recurrenceRules: null
```

```
            }
        ];

        return events;
    }

    getTodayTimestampAtTimeOfDay(hours, minutes, seconds) {
        let d = new Date();
        d.setHours(hours, minutes, seconds, 0);
        return d.getTime() / 1000; // milliseconds to seconds
    }

    timeOfDayToString(dateSecondsUTC) {
        let d = new Date(dateSecondsUTC * 1000); // seconds to milliseconds
        let ampm = "AM";
        let str = "";

        if (d.getHours() > 12) {
            ampm = "PM";
            str += `${d.getHours() - 12}`;
        } else {
            str += `${d.getHours()}`;
        }

        if (d.getMinutes() > 0) {
            if (d.getMinutes() < 10) {
                str += `:0${d.getMinutes()}`;
            } else {
                str += `:${d.getMinutes()}`;
            }
        }

        str += ` ${ampm}`;

        return str;
    }

    alarmsToString(alarms) {
        let results = [];

        if (alarms) {
            alarms.forEach(alarm => {
                if (alarm.relativeOffsetSeconds == 0) {
                    results.push("At time of event");
                } else {
                    const mins = parseInt(Math.ceil(Math.abs(alarm.relativeOffsetSeconds)
/ 60.0));
                    if (mins == 1) {
                        results.push("1 minute before");
                    } else if (mins < 60) {
                        results.push(`${mins} minutes before`);
                    } else {
                        const hours = parseInt(Math.ceil(mins / 60.0));
                        if (hours == 1) {
```

```
                            results.push("1 hour before");
                        } else if (hours < 24) {
                            results.push(`${hours} hours before`);
                        } else {
                            const days = parseInt(Math.ceil(hours / 24.0));
                            if (days == 1) {
                                results.push("1 day before");
                            } else {
                                results.push(`${days} days before`);
                            }
                        }
                    }
                }
            });
        } else {
            results.push("None");
        }

        return results;
    }

    generateDisplayFields(calendarEvent) {
        // these are used for display purpose only
        calendarEvent.startTimeDisplay =
this.timeOfDayToString(calendarEvent.startDateSecondsUTC);
        calendarEvent.endTimeDisplay =
this.timeOfDayToString(calendarEvent.endDateSecondsUTC);
        calendarEvent.alarmsDisplay = this.alarmsToString(calendarEvent.alarms);
    }

    showSuccessAlert(title, message) {
        LightningAlert.open(
            {
                message: message,
                theme: "success",
                label: title,
            }
        );
    }

    showFailureAlert(title, message) {
       console.log(`calendarService.isAvailable(): ${this.calendarService.isAvailable()}`);

        console.log(`selectedItemIndex: ${this.selectedItemIndex}`);
        console.log(`selectedItem: ${this.selectedItem.id}`);

        LightningAlert.open(
            {
                message: message,
                theme: "error", // a red theme intended for error states
                label: title,
            }
        );
    }
```

```
}
```

SEE ALSO:

Use the CalendarService API

Compatibility and Requirements

CalendarService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It requires access to device hardware and device platform APIs. This access is only available when CalendarService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

CalendarService is available in Lightning apps distributed using:

- Mobile Publisher for Experience Cloud

CalendarService is fully functional when used in a Lightning app run from a compatible Salesforce mobile app on a compatible iOS or Android mobile device. See Requirements for Mobile Publisher for Experience Cloud or the requirements page for your target mobile app for specific device and operating system requirements.

CalendarService is **not** fully functional when running on other devices, such as a desktop, or when running in a web browser, even on a mobile device. It **requires** one of the mobile apps listed above. The CalendarService API is accessible in Lightning Experience on all devices, so your code won't fail due to missing functions. However, when your app runs in a browser–desktop or mobile–it can use only CalendarService constants and utility functions. Attempting any calendar-related operation will fail.

Considerations and Limitations

Keep the following in mind when developing features that use the CalendarService API.

Device Limitations

CalendarService doesn't manage calendar and event data itself. Instead, it makes available certain calendar and event data of the underlying platform (Android or iOS) and hardware (phone or other mobile device). While the features provided by CalendarService are the same across both platforms, they're subject to some platform-specific quirks and minor differences.

- In order for CalendarService to access a calendar and perform associated actions, the account associated with a calendar must be synced to the device and grant the device permission to access the calendar. For example, if a user has an email account synced to their device but has not granted the device access to that account's calendar, CalendarService won't be able to interact with that calendar.
- Some devices have other restrictions, such as an employer-managed MDM, that limit access to certain device calendar APIs. Such restrictions can prohibit CalendarService from accessing and interacting with these calendars.
- CalendarService requires the use of the mobile device's calendar. Your user must grant your app access to the calendar. The exact user experience is governed by the platform. The request happens automatically on first use, and is managed by the device itself, but you should plan for it when designing the user experience of your app.
- In Android 11 or later, if the user taps "Deny" for permission to access the Calendar app more than once during the app's lifetime of installation on a device, the user won't see the system permissions dialog again. Tapping Deny multiple times implicitly chooses the "don't ask again" option.

In previous versions of Android, users would see the system permissions dialog each time the app requested permission unless the user had previously selected "don't ask again". This change in Android 11 discourages repeated requests for permissions that users have chosen to deny.

If the user has denied permission to access the Calendar app and needs to change their permissions to allow access, they can do so in their device's settings.

Development Considerations

You can certainly develop the user experience for your component on a desktop or laptop development system. But be sure to test calendar and event functionality on the physical devices on which you plan to deploy your Lightning app.

Important: CalendarService allows for actions that, if used irresponsibly or incorrectly, can lead to irreversible consequences on your users' devices. These **dangerous actions** include altering and deleting calendar events.

As with all mobile capabilities, implementation of CalendarService's functionality, dangerous actions included, is at your discretion. Use caution when using dangerous actions in your component.

CalendarService Considerations

CalendarService is built on top of mobile operating system and device features. CalendarService's capabilities therefore depend on Android or iOS features, which are subject to change beyond our control. When mobile operating system features change, the behavior of CalendarService can change without notice.

Access Contacts on a Mobile Device

A Lightning web component can use a mobile device's contacts feature to select contacts from the device. When contacts are selected, they're returned to the Lightning web component that invoked it. Your component can import the contacts into Salesforce, attach contact data to a record, or otherwise process the contacts as needed.

Contacts are accessed locally on the mobile device, and ContactsService doesn't require a network connection. However, ContactsService requires access to platform-specific APIs that are available only within compatible Salesforce mobile apps.

Important: ContactsService does not and cannot function when running in a web browser, whether on a desktop or mobile device.

IN THIS SECTION:

ContactsService User Experience

Your component can deliver any user experience you desire, but there's a common flow for any component that imports or processes contact data.

Use the ContactsService API

To develop a Lightning web component with contacts-based features, use the ContactsService API as your method for selecting contacts from a device's address book.

ContactsService Example

Here's a minimal but complete example of a Lightning web component that uses ContactsService to select on-device contacts and then process the contact data in the component.

Compatibility and Requirements

ContactsService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when ContactsService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

Considerations and Limitations
Keep the following in mind when developing features that use the ContactsService API.

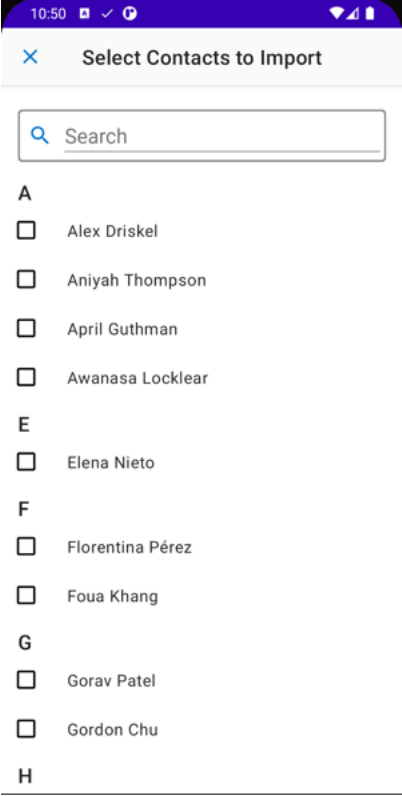
SEE ALSO:

Lightning Web Components Developer Guide: ContactsService API

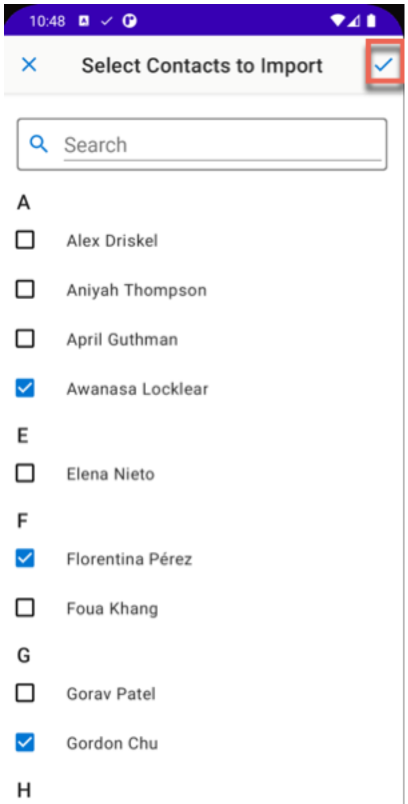
ContactsService User Experience

Your component can deliver any user experience you desire, but there's a common flow for any component that imports or processes contact data.

1. A Lightning web component displays a button (or other user interface control) to start a process that uses contacts data.
2. When the button is tapped, ContactsService opens a list view for the user to select any number of contacts from the device's native contacts list.



3. After the desired contacts are selected, tap the "check" icon or **Done** button to continue.



4. Your Lightning web component receives the data from the selected contacts. Your component can display additional user interface controls to further process the contacts, add the contacts to Salesforce, or otherwise apply whatever custom logic your business process requires.

Use the ContactsService API

To develop a Lightning web component with contacts-based features, use the ContactsService API as your method for selecting contacts from a device's address book.

1. Import ContactsService into your component definition to make the ContactsService API functions available to your code.
2. Test to make sure ContactsService is available before you call contacts-related functions.
3. Use the `getContacts()` function to select and access contacts.

Add ContactsService to a Lightning Web Component

In your component's JavaScript file, import ContactsService using the standard JavaScript `import` statement. Specifically, import the `getContactsService()` factory function from the `lightning/mobileCapabilities` module, like so:

```
import { getContactsService } from 'lightning/mobileCapabilities';
```

After it's imported into your component, use the factory function to get an instance of ContactsService. With your ContactsService instance, use the utility functions and constants to verify availability. Then use contacts-related functions to select contacts from the device.

Test ContactsService Availability

ContactsService depends on physical device hardware and platform features. A component that uses ContactsService renders without errors on a desktop computer or in a mobile browser, but contacts-related functions fail. To avoid these errors, check if ContactsService functionality is available before you use it.

```
handleGetContactsClick(event) {
    const myContactsService = getContactsService();
    if(myContactsService.isAvailable()) {
        // Perform contacts-related operations
    }
    else {
        // ContactsService not available, or consuming app hasn't implemented it
        // Not running on hardware with contacts, address book, etc.
        // Handle with message, error, beep, and so on
    }
}
```

Access and Process Contacts

It's straightforward to create a custom contacts processing feature using ContactsService.

1. Open the contacts selection list view using `getContacts()`.
2. Your user selects contacts from the list.
3. Process the results (the contacts data for the selected contacts).

For example:

```
// Select on-device contacts, and then process them
myContactsService.getContacts(options)
.then((results) => {
    // Do something with the contacts data
    this.contacts = results;
    console.log(results);
})
.catch((error) => {
    // Handle cancellation, or selection errors here
    this.contacts = [];
    console.error('Error code: ' + error.code); +
    console.error('Error message: ' + error.message);
});
```

See `getContacts(options)` for more details about how to handle contacts data, handle errors, and so on.

SEE ALSO:

Lightning Web Components Developer Guide: ContactsService API

ContactsService Example

ContactsService Example

Here's a minimal but complete example of a Lightning web component that uses ContactsService to select on-device contacts and then process the contact data in the component.

The component's HTML template is minimal, with a button to start selecting contacts, and a place to display results.

```
<!-- contactsServiceExample.html -->
<template>

    <!-- User interface control -->
    <lightning-card title="ContactsService Example">
        <div class="slds-var-m-around_medium">
            <lightning-button
                variant="brand"
                label="Process Contacts"
                onclick={handleImportContacts}
                class="slds-var-m-left_x-small">
            </lightning-button>
        </div>
    </lightning-card>

    <!-- After contacts are selected, they're displayed here -->
    <div class="slds-var-m-around_medium">
        <p><b>Contacts Selected</b></p>
        <p><lightning-formatted-text
            value={contactsResults}></lightning-formatted-text></p>
        <ul class="slds-var-m-around_medium">
            <template for:each={contacts} for:item="contact">
                <li key={contact.Id}>
                    {contact.name.givenName} {contact.name.familyName}
                    <ul class="slds-var-m-around_medium">
                        <template for:each={contact.phoneNumbers}
                            for:item="phoneNumber">
                            <li key={phoneNumber.value}>
                                {phoneNumber.value} ({phoneNumber.label})
                            </li>
                    </template>
                    </ul>
                </li>
            </template>
        </ul>
      </div>

</template>
```

Note the use of the `for:each` directive and nested `template` tag to iterate through the list of contacts, and a further nested `for:each` directive and template to iterate through each contact's list of phone numbers. You can do the same for email addresses, IM handles, and other contact details.

This example doesn't import contacts into Salesforce, or otherwise use the contacts data provided from the device. It simply uses ContactsService to allow you to select contacts on the device, and then displays the returned contact data, or a status message when there's an error.

```
// contactsServiceExample.js
import { LightningElement } from 'lwc';
import { getContactsService } from 'lightning/mobileCapabilities';

export default class ContactsServiceExample extends LightningElement {

    // Component state: result status, and result contacts data
    contactsResults = 'No contacts selected.';
    contacts = [];

    handleImportContacts() {

        const myContactsService = getContactsService();

        // Make sure ContactsService is available before trying to access contacts
        if(myContactsService.isAvailable()) {

            // Configuration for ContactsService
            let options = {
                "permissionRationaleText":
                    "Allow access to your contacts to enable contacts processing."
            };

            // Select on-device contacts, and then process them
            myContactsService.getContacts(options)
            .then((results) => {
                this.contacts = results;
                this.contactsResults = 'Number of contacts selected: ' +
                    this.contacts.length;
            })
            .catch((error) => {
                // Handle cancellation, or selection errors here
                this.contacts = [];
                this.contactsResults = JSON.stringify(error) +
                    '\n\nError code: ' + error.code +
                    '\n\nError message: ' + error.message;
                console.error(this.contactsResults);
            })
        } else {
            // ContactsService isn't available
            // Are you running in a supported mobile app?
            this.contactsResults = "ContactsService API isn't available.";
        }
    }
```

```
}
```

SEE ALSO:

Use the ContactsService API

Lightning Web Components Developer Guide: Render Lists

Compatibility and Requirements

ContactsService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when ContactsService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

ContactsService is available in Lightning apps distributed using:

- Mobile Publisher for Salesforce App
- Mobile Publisher for Experience Cloud

ContactsService is fully functional when used in a Lightning app or Lightning site run from a compatible Salesforce mobile app on a compatible iOS or Android mobile device. See Requirements for the Salesforce Mobile App, Requirements for Mobile Publisher for Salesforce App, or the requirements page for your target mobile app for specific device and operating system requirements.

ContactsService is **not** fully functional when running on other devices, such as a desktop, or when running in a web browser, even on a mobile device. It **requires** one of the mobile apps listed above. The ContactsService API is accessible in Lightning Experience on all devices, so your code won't fail due to missing functions. However, when your app runs in a browser—desktop or mobile—it can use only ContactsService constants and utility functions. Attempting any contacts-related operation will fail.

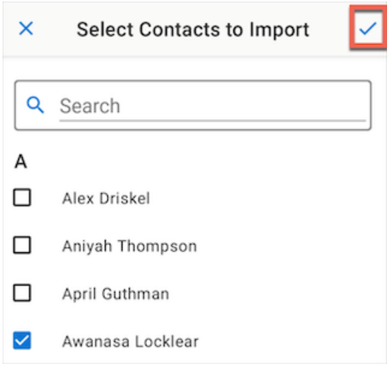
Considerations and Limitations

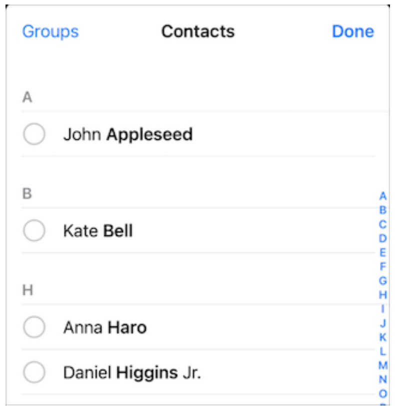
Keep the following in mind when developing features that use the ContactsService API.

Device Limitations

ContactsService doesn't manage contact information itself. Instead, it makes available certain contact data of the underlying platform (Android or iOS) and hardware (phone or other mobile device). While the features provided by ContactsService are the same across both platforms, they're subject to some platform-specific quirks and minor differences.

- The user interface for selecting contacts are subject to differences between platforms. For example, the **Done** button on iOS vs. a checkmark icon on Android, and differences in cancellation:





- ContactsService requires the use of the mobile device's contacts. Your user must grant your app access to contacts. The exact user experience is governed by the platform. The request happens automatically on first use, and is managed by the device itself, but you should plan for it when designing the user experience of your app.
- In Android 11 or later, if the user taps "Deny" for permission to access the Contacts app more than once during the app's lifetime of installation on a device, the user won't see the system permissions dialog again. Tapping Deny multiple times implicitly chooses the "don't ask again" option.

  In previous versions of Android, users would see the system permissions dialog each time the app requested permission unless the user had previously selected "don't ask again". This change in Android 11 discourages repeated requests for permissions that users have chosen to deny.

  If the user has denied permission to access the Contacts app and needs to change their permissions to allow access, they can do so in their device's settings.

- If Mobile Device Management (MDM) is in use on a device, and MDM restricts access to contacts data, ContactsService can't access it.
- If the user doesn't have any contacts in their device's contact list, ContactsService won't work.
- If the user has contacts associated with specific accounts on their device, those accounts must have contacts sync enabled in order for ContactsService to access them. In other words, ContactsService can only access contacts that are visible in your user's device's native contact list. See Use other contact accounts on iPhone (iOS) and Back up & sync device contacts (Android) for more information.

If you're having trouble getting ContactsService to access contact data, try the following:

- First, verify that your device contact list contains the contact data you're trying to access. ContactsService can only access contacts that are visible in your device's native contact list.
- Next, verify that you've granted contacts permission to the mobile app where your component is running. You can check this in your device's settings.
- Finally, double-check the configuration of ContactsService in your code.

Development Considerations

Virtual devices might not have any contact data when they're created. To test ContactsService on a virtual device, you'll first need to create some contact records in the virtual device's Contacts app. Alternatively, you can import contact data in the standard .vcf format, or sign into a real Google or iCloud account that has contact data associated with it.

You can certainly develop the user experience for your component on a desktop or laptop development system. But be sure to test contact access functionality on the physical devices on which you plan to deploy your Lightning app.

ContactsService Considerations

Be aware of the following considerations when using ContactsService in your Lightning app.

- ContactsService is built on top of mobile operating system and device features. ContactsService's capabilities therefore depend on Android or iOS features, which are subject to change beyond our control. When mobile operating system features change, the behavior of ContactsService can change without notice.
- A Lightning component that uses ContactsService can have a custom user interface in the component itself. However, the contacts selection user interface can't be customized.

Scan Documents on a Mobile Device

A Lightning web component can use a device's camera and optical character recognition (OCR) to scan documents. When a document is successfully scanned, text data extracted from the scanned document is returned to the Lightning web component that invoked it. DocumentScanner recognizes printed text and handwritten form factors. However, DocumentScanner provides the most accurate results when scanning printed text compared handwritten text, which varies on the legibility of handwritten characters.

DocumentScanner results are returned in two formats:

- A simple string of all recognized text, suitable for simple document capture.
- Structured text data aligned with the scanned image, suitable for interactive processing of document content.

Scanning is performed locally on the mobile device, and doesn't need a network connection. However, DocumentScanner requires access to APIs implemented in platform-native code that are available only within compatible Salesforce mobile apps.

Important: DocumentScanner does not and cannot function when running in a web browser, whether on a desktop or mobile device.

IN THIS SECTION:

Use the DocumentScanner API

To add document scanning features to a Lightning web component, use the DocumentScanner API.

DocumentScanner Example

Here's an example of a Lightning web component that uses DocumentScanner to capture text data from an image.

Compatibility and Requirements

DocumentScanner is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when DocumentScanner runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

Considerations and Limitations

Consider these guidelines and limitations when developing features that use the DocumentScanner API.

Use the DocumentScanner API

To add document scanning features to a Lightning web component, use the DocumentScanner API.

1. Import DocumentScanner into your component definition to make the DocumentScanner API functions available to your code.
2. Test to make sure DocumentScanner is available before you call the `scan()` function.
3. Use the `scan()` function to begin a document scanning session.
4. Process the scan results.

For complete reference documentation of the DocumentScanner API, see DocumentScanner API in the *Lightning Web Components Developer Guide*.

Add DocumentScanner to a Lightning Web Component

In your component's JavaScript file, import DocumentScanner using the standard JavaScript `import` statement. Specifically, import the `getDocumentScanner()` factory function from the `lightning/mobileCapabilities` module, like so:

```
import { getDocumentScanner } from 'lightning/mobileCapabilities';
```

After it's imported into your component, use the factory function to get an instance of DocumentScanner. With your DocumentScanner instance, use the utility functions and constants to verify availability. Then use the scan function to initiate a document scan.

Test DocumentScanner Availability

DocumentScanner depends on physical device hardware and platform features. A component that uses DocumentScanner renders without errors on a desktop computer or in a mobile browser, but scanning functions fail.. To avoid these errors, test if DocumentScanner functionality is available before you use it.

```
handleBeginScanEventClick(event) {
    const myDocumentScanner = getDocumentScanner();
    if(myDocumentScanner.isAvailable()) {
        // Perform document scanning operations
    }
    else {
        // DocumentScanner not available, or consuming app hasn't implemented it
        // Not running on hardware with a scanner
        // Handle with message, error, beep, and so on
    }
}
```

Scan a Document

Scanning documents with DocumentScanner is straightforward.

- Start a scan with `scan(options)`.
- (Your user scans a document.)
- Handle the result of the scan, which is returned in the form of a promise.

For example:

```
myScanner
  .scan(scanningOptions)
  .then((result) => {
    // Do something with the result of the scan
    console.log(result);
    this.scannedDocument = result.value;
  })
  .catch((error) => {
    // Handle cancellation and scanning errors here
    console.error(error);
  });
```

When scanning is successful, the returned promise contains the string value of the scanned text. See (reference doc link) for the structure and contents of the result.

If scanning is unsuccessful, the promise is returned as an error, which includes an error code and message that you can use in error handling. See DocumentScanner API in the *Lightning Web Components Developer Guide* for more information.

Process Scan Results

When scanning completes successfully, DocumentScanner returns an array of Document objects in the result object. Document objects include scan data in two different formats.

- A simple string of all recognized text.
- Structured text data that includes positional details aligned with the underlying scanned image.

Both formats are available from Documents in the result object. The simple string of all recognized text makes it easy to implement simple features where you scan a document into a text field. The structured text results require more processing to handle, but allow you to develop complex UIs for extracting portions of the scanned document, assign elements to multiple form fields, and so on. See DocumentScanner Example for an example of handling both types of scanned document data.

DocumentScanner Example

Here's an example of a Lightning web component that uses DocumentScanner to capture text data from an image.

There's an option to select the source of the image to be scanned, from either the device camera or the device image library.

```
<template>
  <table class="rootTable">
   <tbody>

    <!-- Document scanning controls -->
    <tr>
     <td style="height: 1px;">
      // Choose source of the document to be scanned
      <lightning-card title="Document Scanner" icon-name="custom:display_text">
       <div class="slds-var-p-around_medium">
        Select source of document to be scanned:
        <br/><br/>

        <lightning-button
         variant="brand"
         label="Camera"
         title="Capture document with camera"
         onclick={handleScanFromCameraClick}>
        </lightning-button>

        <lightning-button
         variant="brand"
         label="Photo Library"
         title="Scan document from photo library"
         onclick={handleScanFromPhotoLibraryClick}
         class="slds-var-m-left_x-small">
        </lightning-button>
       </div>
```

```
        <!-- Display errors, if any -->
        <template if:true={scannerError}>
         <lightning-formatted-text value={scannerError}>
         </lightning-formatted-text>
        </template>

        <!-- Display text of scanned document, if any -->
        <template if:true={scannedDocument}>
         // results of the scan are displayed here
         <div class="slds-var-p-around_medium">
          Text Recognition Result: <br/><br/>
          {scannedDocument.text}
         </div>
        </template>

      </lightning-card>
     </td>
    </tr>

    <!-- If there is a scanned document, display a preview -->
    <tr>
     <td>
      <template if:true={scannedDocument}>
       <div class="previewDiv">

        <!-- document image -->
        <div class="divContentCentered">
         <img class="previewImage" src={scannedDocument.imageBytes}
          onload={addImageHighlights} />
        </div>

        <!-- highlights overlay; note use of manual DOM rendering -->
        <div class="divContentCentered">
         <div class="contour" lwc:dom="manual"></div>
        </div>

       </div>
      </template>
     </td>
    </tr>

   </tbody>
  </table>
 </template>
```

This example uses DocumentScanner to choose a source for the document to be scanned, and to perform a basic scanning operation. After the scan completes, the results are displayed, along with an SVG graphic that annotates the scanned document graphic with an overlay of the result structured text data. A status message is returned when there's an error.

```
import { LightningElement } from "lwc";
import { getDocumentScanner } from "lightning/mobileCapabilities";

export default class DocumentScanner extends LightningElement {
  // Scan results (if any)
  scannerError;
```

```
scannedDocument;

handleScanFromCameraClick() {
  this.scanDocument("DEVICE_CAMERA");
}

handleScanFromPhotoLibraryClick() {
  this.scanDocument("PHOTO_LIBRARY");
}

scanDocument(imageSource) {
  // Clear previous results / errors
  this.resetScanResults();

  // Main document scan cycle
  const myScanner = getDocumentScanner();
  if (myScanner.isAvailable()) {
    // Configure the scan
    const options = {
      imageSource: imageSource,
      scriptHint: "LATIN",
      returnImageBytes: true,
    };

    // Perform document scan
    myScanner
      .scan(options)
      .then((results) => {
        // Do something with the results
        this.processScannedDocuments(results);
      })
      .catch((error) => {
        // Handle errors
        this.scannerError =
          "Error code: " + error.code + "\nError message: " + error.message;
      });
  } else {
    // Scanner not available
    this.scannerError =
      "Problem initiating scan. Are you using a mobile device?";
  }
}

resetScanResults() {
  this.scannedDocument = null;
  this.scannerError = null;
}

processScannedDocuments(documents) {
  // DocumentScanner only processes the first scanned document in an array
  this.scannedDocument = documents[0];
  // And this is where you take over; process results as desired
}
```

```
  // Build an annotation overlay graphic, to display on top of the scanned image
  addImageHighlights(event) {
    const textBlocks = this.scannedDocument?.blocks;
    if (!textBlocks) {
      return;
    }

    const img = event.srcElement;
    const cWidth = img.clientWidth;
    const cHeight = img.clientHeight;
    const nWidth = img.naturalWidth;
    const nHeight = img.naturalHeight;
    const width = Math.min(cWidth, nWidth);
    const height = Math.min(cHeight, nHeight);

    let svg =
      `<svg version="1.1" xmlns="http://www.w3.org/2000/svg" ` +
      `xmlns:xlink="http://www.w3.org/1999/xlink" ` +
      `width="${width}" height="${height}" viewBox="0, 0, ${nWidth}, ${nHeight}">`;
    textBlocks.forEach((block) =>
      block.lines.forEach((line) =>
        line.elements.forEach((element) => {
          const frame = element.frame;
          svg +=
            `<rect x="${frame.x}" y="${frame.y}" width="${frame.width}" ` +
            `height="${frame.height}" style="fill:green;fill-opacity:0.5" />`;
        })
      )
    );
    svg += "</svg>";

    // Manually attach the overlay SVG to the LWC DOM to render it
    this.template.querySelector(".contour").innerHTML = svg;
  }
}
```

Compatibility and Requirements

DocumentScanner is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when DocumentScanner runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

DocumentScanner is available in Lightning apps distributed using:

- Salesforce Mobile app
- Salesforce Field Service app

DocumentScanner is fully functional when used in a Lightning app or Lightning site that's run from a compatible Salesforce mobile app on a compatible iOS or Android mobile device. See Requirements for the Salesforce Mobile App, or the requirements page for your target mobile app for specific device and operating system requirements.

DocumentScanner is **not** fully available when running on other devices, such as a desktop, or when running in a web browser, even on a mobile device. It **requires** one of the apps listed above. The DocumentScanner API is accessible in Lightning Experience on all devices, so your code won't fail due to missing functions. However, when your app runs in a browser—desktop or mobile—it can use only DocumentScanner constants and utility functions. Attempting any document scanning operation will fail.

Considerations and Limitations

Consider these guidelines and limitations when developing features that use the DocumentScanner API.

Device Limitations

- To scan documents using a mobile device's camera, DocumentScanner requires permission to use the camera. Your user must grant your app access to the camera. The exact user experience is governed by the platform. The request happens automatically on first use, and is managed by the device itself, but you should plan for it when designing the user experience of your app.
- To scan documents stored in the device photo library, DocumentScanner requires permission to access the photo library. Your user must grant your app access to the photo library. Again, the user experience is managed by the platform (iOS or Android).
- In Android 11 or later, if the user taps "Deny" for permission to access the Camera app or photo library more than once during the app's lifetime of installation on a device, the user won't see the system permissions dialog again. Tapping Deny multiple times implicitly chooses the "don't ask again" option.

  In previous versions of Android, users would see the system permissions dialog each time the app requested permission unless the user had previously selected "don't ask again". This change in Android 11 discourages repeated requests for permissions that users have chosen to deny.

  If the user has denied permission to access the Camera app or photo library and needs to change their permissions to allow access, they can do so in their device's settings.

- DocumentScanner doesn't implement scanning itself. Instead, it delegates document scanning and recognition to an underlying SDK implemented in native code (Android or iOS). While the features provided by DocumentScanner are the same across both platforms, they're subject to some platform-specific quirks and minor differences.
  - If the chosen input for the document to be scanned is the device camera, DocumentScanner automatically uses the "main" device camera. Built-in controls allow choosing other device cameras, if available.
  - If you can't get a clear picture of the document, it can't be recognized. The quality of the device camera affects text recognition. A damaged or low-quality camera lens or focusing system, poor lighting, motion, and other factors can make it difficult or impossible to get a clear picture of a document.
  - The quality of the document affects text recognition. Specifically, documents with damaged or obscured text are hard to recognize successfully.
  - DocumentScanner recognizes printed text and handwritten form factors. However, DocumentScanner provides the most accurate results when scanning printed text compared handwritten text, which varies on the legibility of handwritten characters. While DocumentScanner is capable of performing text recognition on a variety of typefaces, highly stylized fonts can lower recognition accuracy.

Development Considerations

- DocumentScanner requires access to camera hardware, or to the device photo library. To test scanning using a camera during development, use actual, physical devices.
  - The Android emulator can simulate camera hardware by using a webcam on your development system. To do so, edit the camera configuration for your Android Virtual Device, in the advanced settings panel. However, the camera built into most laptops is much lower quality than what's found on modern mobile phones. A low-quality camera limits the usefulness of testing text recognition.

  - The iOS simulator doesn't provide access to simulated camera hardware at all. You can certainly develop the user experience for your component on a desktop or laptop development system. But, be sure to test camera scanning functionality on the physical devices on which you plan to deploy your Lightning app.

- As a work-around for limited camera access in virtual devices, you can load a few sample images into the virtual device's photo library, and use those for scanning operations. However, if you expect your users to scan using a device camera, be sure to also test using real device cameras.

DocumentScanner Considerations

DocumentScanner is built on top of mobile operating system and device features. DocumentScanner's capabilities therefore depend on Android or iOS features, which are subject to change beyond our control. When mobile operating system features change, the behavior of DocumentScanner can change without notice.

Monitor Geofence Regions on a Mobile Device

A Lightning web component can use a mobile device's location features to determine a user's current location to the user's proximity to areas that may be of interest, or to perform location-related tasks. The longitude, latitude, and radius define a geofence around the regions of interest.

Geofence location is determined locally on the mobile device, and doesn't need a network connection. However, GeofencingService does require a GPS signal from the device. For Android devices, Google Location Accuracy must be enabled in the system settings. GeofencingService does require access to platform-specific APIs that are available only within compatible Salesforce mobile apps.

Important: GeofencingService does not and cannot function when running in a web browser, whether on a desktop or mobile device.

IN THIS SECTION:

GeofencingService User Experience

Your component can deliver any user experience you desire. There are a number of geofencing-based features where GeofencingService might be suitable.

Use the GeofencingService API

To develop a Lightning web component with location-based features for creating and monitoring geofences, use the GeofencingService API.

GeofencingService Example

Here's a basic example of a Lightning web component that uses a device's biometrics capabilities to verify device ownership.

Compatibility and Requirements

GeofencingService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when GeofencingService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

Considerations and Limitations

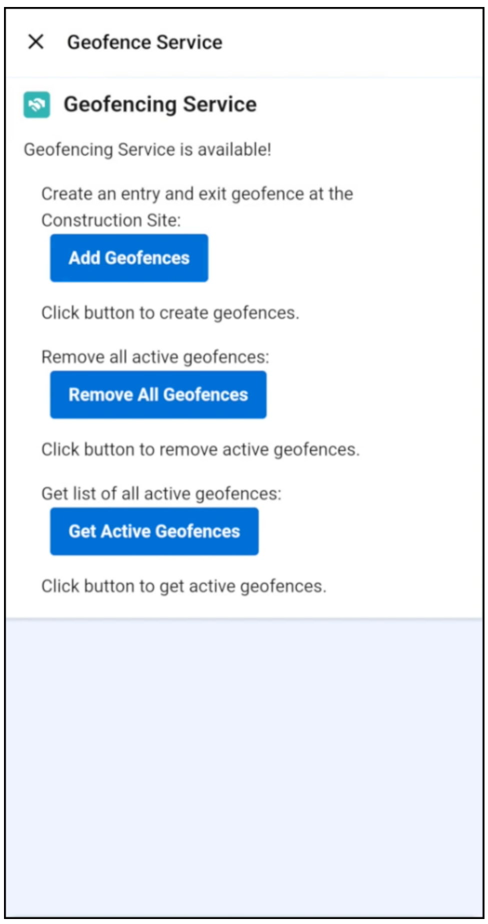
Consider these guidelines and limitations when developing features that use the GeofencingService API.

SEE ALSO:

Lightning Web Components Developer Guide: GeofencingService API

GeofencingService User Experience

Your component can deliver any user experience you desire. There are a number of geofencing-based features where GeofencingService might be suitable.



- Your user enters or exits a geofence region and triggers an event.
- Your user enters an area of interest and receives a notification.
- Display an alert based on user location.

Use the GeofencingService API

To develop a Lightning web component with location-based features for creating and monitoring geofences, use the GeofencingService API.

1. Import GeofencingService to make the GeofencingService API functions available to your code.
2. Test to make sure GeofencingService is available before you call geofencing functions.
3. Use the geofencing functions to get the current location and to define regions of interest.

Add GeofencingService to a Lightning Web Component

In your component's JavaScript file, import GeofencingService using the standard JavaScript `import` statement. Specifically, import the `getGeofencingService()` factory function from the `lightning/mobileCapabilities` module, like so:

```
import { getGeofencingService } from 'lightning/mobileCapabilities';
```

After it's imported into your component, use the factory function to get an instance of GeofencingService. With your GeofencingService instance, use the utility functions and constants to verify availability. Then use the geofencing function to create location arrival or departure notifications for geographic regions.

Test GeofencingService Availability

GeofencingService depends on physical device hardware and platform features. A component that uses GeofencingService renders without errors on a desktop computer or in a mobile browser, but geofencing functions fail. To avoid these errors, test whether GeofencingService functionality is available before you use it.

```
handleClickAddGeofencingRegion(event) {
    const myGeofencingService = getGeofencingService();
    if(myGeofencingService.isAvailable()) {
        // Perform geofencing operations
    }
    else {
        // GeofencingService not available
        // Not running on hardware with location APIs, etc.
        // Handle with message, error, beep, and so on
    }
}
```

Start Monitoring a Geofence Region

Use the `StartMonitoringGeofence()` function to start monitoring a geofence region. Then, handle the outcome in whatever manner you wish.

```
sftowerEntry = {
    latitude: 37.7899,
    longitude: -122.3969,
    radius: 50,
    notifyOnEntry: true,
    message: “Welcome to Salesforce Tower”,
    triggerOnce: false
}

addGeofence() {
    // …
    if(myGeofencingService.isAvailable()) {
        // Perform geofencing operations
        myGeofencingService.startMonitoringGeofence(this.sftowerEntry)
             .then((id) => {
                 console.log(`ID for geofence SF Tower Entry created: ${id}`);
             })
             .catch((error) => {
                 console.log(error);
             });
```

```
    }
}
```

Stop and Remove All Monitored Geofence Regions

Use the `StopMonitoringAllGeofences()` function to stop monitoring all geofence regions. Callback is called when the monitoring stops for all geofence regions, or with an error if the monitoring fails to stop.

```
handleClickRemoveGeofencingRegion(geofenceRegionId) {
    // …
    if(myGeofencingService.isAvailable()) {
        // Perform geofencing operations
        myGeofencingService.stopMonitoringGeofence(geofenceRegionId)
             .then(() => {
                 console.log(`Success`);
             })
             .catch((error) => {
                 console.log(error);
             });
    }
}
```

Stop and Remove A Specific Monitored Geofence Region

Use the `StopMonitoringAllGeofences()` function to stop monitoring all geofence regions. Callback is called when the monitoring stops for a geofence region, or with an error if the monitoring fails to stop.

```
handleClickRemoveGeofencingRegion(geofenceRegionId) {
    // …
    if(myGeofencingService.isAvailable()) {
        // Perform geofencing operations
        myGeofencingService.stopMonitoringGeofence(geofenceRegionId)
             .then(() => {
                 console.log(`Success`);
             })
             .catch((error) => {
                 console.log(error);
             });
    }
}
```

Get the IDs of Active Geofences

Use the `getMonitoredGeofences()` function to get the IDs of active geofence regions. Provide a callback function to handle all IDs of geofences being monitored, or with an error if the query fails.

```
handleClickGetAllGeofenceIDs() {
    // …
    if(myGeofencingService.isAvailable()) {
        // Perform geofencing operations
        myGeofencingService.getMonitoredGeofences()
             .then((results) => {
```

```
                const activeGeofences = JSON.parse(JSON.stringify(results));
                const msg = `Number of geofences: + ${activeGeofences.length}`;
              console.log(msg);
          })
          .catch((error) => {
              console.log(error);
          });
    }
}
```

SEE ALSO:

Lightning Web Components Developer Guide: GeofencingService API

GeofencingService Example

GeofencingService Example

Here's a basic example of a Lightning web component that uses a device's biometrics capabilities to verify device ownership.

Here's a basic example of a Lightning web component that uses GeofencingService to monitor and determine when a user arrives or departs a geographic region.

```
<template>
   <lightning-card title="Geofencing Service" icon-name="custom:custom14">
       <div class="slds-var-m-around_medium">
           <p><lightning-formatted-text
value={geofencingResults}></lightning-formatted-text></p>
           <div class="slds-var-m-around_medium">
               <p>Create an entry and exit geofence at the Salesforce Tower:</p>
               <lightning-button variant="brand" label="Add Geofences"
                   title="Add Geofences to SF Tower"
                   onclick={addGeofence}
                   class="slds-var-m-around_x-small"
                   disabled={geofencingServiceDisabled}>
               </lightning-button>
           </div>
           <div class="slds-var-m-around_medium">
               <p><lightning-formatted-text
value={geofencingAddedResults}></lightning-formatted-text></p>
           </div>
       </div>
       <div class="slds-var-m-around_medium">
           <div class="slds-var-m-around_medium">
               <p>Remove all active geofences:</p>
               <lightning-button variant="brand" label="Remove All Geofences"
                   title="Remove all geofences"
                   onclick={removeGeofences}
                   class="slds-var-m-around_x-small"
                   disabled={geofencingServiceDisabled}>
               </lightning-button>
           </div>
           <div class="slds-var-m-around_medium">
               <p><lightning-formatted-text
value={removeGeofencesResults}></lightning-formatted-text></p>
```

```
            </div>
        </div>
        <div class="slds-var-m-around_medium">
            <div class="slds-var-m-around_medium">
                <p>Get list of all active geofences:</p>
                <lightning-button variant="brand" label="Get Active Geofences"
                    title="Get active geofences"
                    onclick={getActiveGeofences}
                    class="slds-var-m-around_x-small"
                    disabled={geofencingServiceDisabled}>
                </lightning-button>
            </div>
            <div class="slds-var-m-around_medium">
                <p><lightning-formatted-text
value={activeGeofencesResults}></lightning-formatted-text></p>
                <ul class="slds-var-m-around_medium">
                    <template for:each={activeGeofences} for:item="geofence">
                        <li key={geofence}>{geofence}</li>
                    </template>
                </ul>
            </div>
        </div>
    </lightning-card>
</template>
```

SEE ALSO:

Use the GeofencingService API

Compatibility and Requirements

GeofencingService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when GeofencingService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

GeofencingService is available in Lightning apps distributed using:

- Salesforce Field Service app

GeofencingService is fully functional when used in a Lightning app or Lightning site that's run from a compatible Salesforce mobile app on a compatible iOS or Android mobile device. See Field Service Mobile App Requirements, or the requirements page for your target mobile app for specific device and operating system requirements.

GeofencingService is **not** fully available when running on other devices, such as a desktop, or when running in a web browser, even on a mobile device. It **requires** one of the listed apps. The GeofencingService API is accessible in Lightning Experience on all devices, so your code won't fail due to missing functions. However, when your app runs in a browser—desktop or mobile—it can use only GeofencingService constants and utility functions. Attempting any geofencing-related operation results in failure.

Considerations and Limitations

Consider these guidelines and limitations when developing features that use the GeofencingService API.

Device Limitations

- GeofencingService requires the use of the mobile device location detection hardware. The user must grant your app access to the device's location. The exact user experience is governed by the platform. The request happens automatically on first use, and is managed by the device itself, but you should plan for it when designing the user experience of your app.
- In Android 11 or later, if the user taps "Deny" for permission to access the mobile device location more than once during the app's lifetime of installation on a device, the user won't see the system permissions dialog again. Tapping Deny multiple times implicitly chooses the "don't ask again" option.

  In previous versions of Android, users would see the system permissions dialog each time the app requested permission unless the user had previously selected "don't ask again". This change in Android 11 discourages repeated requests for permissions that users have chosen to deny.

  If the user has denied permission to access the mobile device location and needs to change their permissions to allow access, they can do so in their device's settings.

- Location tracking is subject to significant privacy, processing, and power use restrictions, imposed by the underlying platform (Android or iOS). See important details in the GeofencingService API reference documentation.
- Many factors affect the accuracy of location detection, the speed of determining the current location, and how much impact location tracking has on battery life.
- Mobile devices vary in the quality of the location detection they provide.
- If a mobile device can't determine its location, neither can GeofencingService. The quality of the device's positioning hardware, tall or dense buildings, indoor use, and other external factors can reduce the accuracy of location determination.
- Location tracking on high accuracy can significantly increase power use, which affects how quickly a mobile device drains its battery.

Development Considerations

GeofencingService requires access to positioning hardware, such as GPS, Wi-Fi, cellular, and other location-detection hardware of your mobile device. To test location services during development, use actual, physical devices when possible.

- The Android emulator and iOS simulator can each be configured to provide simulated location details.
- Neither virtual device accurately simulates location detection in the real world, especially for environments where location detection is challenging.

You can certainly develop the user experience for your component on a desktop or laptop development system. But be sure to test geofencing-based functionality on the physical devices on which you plan to deploy your Lightning app.

GeofencingService Considerations

Be aware of the following considerations when using GeofencingService in your Lightning app.

- GeofencingService is built on top of mobile operating system and device features. GeofencingService's capabilities therefore depend on Android or iOS features, which are subject to change beyond our control. When mobile operating system features change, the behavior of GeofencingService can change without notice.
- There's a maximum number of 20 monitored geofences at any time.

Use Location on a Mobile Device

A Lightning web component can use a mobile device's location features to determine the current location of the device and, by association, the person who is holding it. You can access the device's current location at a moment in time, or you can subscribe to location changes, and receive updates to the device's location when it changes significantly.

Location is determined locally on the mobile device, and doesn't need a network connection. LocationService does require access to platform-specific APIs that are available only within compatible Salesforce mobile apps.

Important: LocationService does not and cannot function when running in a web browser, whether running on a desktop or mobile device.

LocationService provides coordinate data only: latitude, longitude, altitude, and some motion details. It doesn't include derived data, such as a physical address or map detail. If you're using the location information to, for example, show a position on a map, you might need a network connection to receive map data, such as map tiles, and so on.

IN THIS SECTION:

LocationService User Experience

Your component can deliver any user experience you desire. There are a number of common location-based features where LocationService might be suitable.

Location Basics

On the surface, the concept of location is a simple one. Where am I? Where is Salesforce Tower? How do I get to Salesforce Tower from where I am right now? These are all location-based features, and we've been using them on mobile devices for many years. As a developer, the concept of location can be more complex.

Use the LocationService API

To develop a Lightning web component with location-based features, use the LocationService API to determine the current location.

LocationService Example

Here's a basic example of a Lightning web component that gets the user's current location and displays it on a map.

Compatibility and Requirements

LocationService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. LocationService **requires** access to device hardware and device platform APIs. This access is only available when LocationService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

Considerations and Limitations

Keep the following in mind when developing features that use the LocationService API.

SEE ALSO:

Lightning Web Components Developer Guide: LocationService API

LocationService User Experience

Your component can deliver any user experience you desire. There are a number of common location-based features where LocationService might be suitable.

Here are a few examples of common location-based features:

- Display a user's current location on a map
- Get the user's current location when they perform an action in your app
- Provide directional guidance for travel between the current location and another location
- Calculate travel time from the current location to another
- Perform an action when a user approaches a destination
- Perform an action when a user moves a significant distance

In these examples, LocationService is usually only a part of the complete solution. Map display, route and travel time calculations, and so on are other parts that you need to implement yourself.

If your feature is a straightforward, map-based page, you can use LocationService position information along with the `lightning-map` base component. For more complex solutions, such as those requiring routing directions, you might need to integrate with other tools or services.

Location Basics

On the surface, the concept of location is a simple one. Where am I? Where is Salesforce Tower? How do I get to Salesforce Tower from where I am right now? These are all location-based features, and we've been using them on mobile devices for many years. As a developer, the concept of location can be more complex.

Location itself can be reduced to basic X,Y coordinates: a latitude and longitude, also known as a *geolocation*. For example, `{37.7898007,-122.3991439}` (the location of Salesforce HQ). But knowing the coordinates of Salesforce Tower doesn't show you where it is on a map, or tell you where you are in relation to it, or how to get there. These are location-based features, but simple location is not enough to build them. It also requires map rendering, geocoding of street addresses, and route calculation, among other things.

LocationService is a simple API, with features inspired by the similar Geolocation API available in web browsers. It provides a straightforward mechanism for getting the current geolocation of the physical device on which it runs. That's it. For basic map display, combine LocationService with the `lightning-map` component. For more complex mapping solutions, consider adding third-party libraries and services, such as Google Maps or Leaflet.

Use the LocationService API

To develop a Lightning web component with location-based features, use the LocationService API to determine the current location.

1. Import LocationService into your component definition to make the LocationService API functions available to your code.
2. Test to make sure LocationService is available before you call location functions.
3. Use the location functions to get the current location, or to request location change updates.

Add LocationService to a Lightning Web Component

In your component's JavaScript file, import LocationService using the standard JavaScript `import` statement. Specifically, import the `getLocationService()` factory function from the `lightning/mobileCapabilities` module, like so:

```
import { getLocationService } from 'lightning/mobileCapabilities';
```

After it's imported into your component, use the factory function to get an instance of LocationService. With your LocationService instance, use the `isAvailable()` utility function to verify availability. Use the location calculation functions to get the current location, or to configure and receive location change updates.

Test LocationService Availability

LocationService depends on physical device hardware and platform features. A component that uses LocationService renders without errors on a desktop computer or in a mobile browser, but location functions fail. To avoid these errors, check if LocationService functionality is available before you use it.

```
handleGetCurrentLocationClick(event) {
    const myLocationService = getLocationService();
    if(myLocationService.isAvailable()) {
        // Perform geolocation operations
    }
```

```
    else {
        // LocationService not available
        // Not running in an app with GPS, location APIs, etc.
        // Handle with message, error, beep, and so on
    }
}
```

Determine Current Location

Determining the current location is a simple function call. While the call is asynchronous, and must be handled as a JavaScript Promise, it's a "one and done" call that allocates and releases resources automatically.

```
myLocationService.getCurrentPosition({ enableHighAccuracy: true }).
    then((result) => {
        this.myLocation = result.coords;
        // Do something with the location here
        // Display a map, look up an address, save to a record
        // □□ It's your thing, do what you wanna do □□
    }).
    catch((error) => {
        // Handle any errors here
        console.error(error);
    });
}
```

See `getCurrentPosition(options)` for more details for configuration, results format, and error handling.

Request Location Change Updates

To receive updates when a device's location changes significantly, subscribe to location updates with the `startWatchingPosition()` function. Provide a callback function to handle position updates when they happen.

```
myLocationWatchId = 0;
async startTracking() {
    let locationService = getLocationService();
    this.myLocationWatchId = await locationService.
        startWatchingPosition({ enableHighAccuracy:true }, (result, error) => {

        // Check for error first
        if(error) {
            console.error(error);
            this.stopTracking();
        }
        else {
            this.myLocation = result.coords;

            // Now do your thing with the updated location
            // Refresh a map, upsert a record, or whatever
        }
    });
}
```

See `startWatchingPosition(options, callback)` for additional usage details, including important resource allocation notes. See `stopWatchingPosition(watchId)` for an example `stopTracking()` function.

SEE ALSO:

Lightning Web Components Developer Guide: LocationService API

LocationService Example

LocationService Example

Here's a basic example of a Lightning web component that gets the user's current location and displays it on a map.

The HTML template provides the bare minimum for a location-based interface. There's an element to display the map, a bit of static help text, and a button to get the location. There are two interesting aspects of this template:

- Disabling the Get Current Location button using the `disabled` attribute when not in a supported Salesforce mobile app. This attribute is set based on the results of `isAvailable()` when the component is initialized.
- A spinner that indicates "indeterminate progress" while waiting for the current location request to resolve.

```
<!-- locationServiceExample.html -->
<template>

   <div class="slds-text-align_center">
       <span class="slds-text-heading_large">Where in the World Am I?</span>
   </div>

   <!-- After the current location is received,
        its value is displayed here: -->
   <template lwc:if={currentLocation}>
       <div class="slds-m-vertical_large slds-p-vertical_medium
                   slds-text-align_left slds-border_top slds-border_bottom">

           <!-- Current location as latitude and longitude -->
           Your current location is:
           <pre>{currentLocationAsString}</pre>

           <!-- Current location as a map -->
           <lightning-map map-markers={currentLocationAsMarker} zoom-level=16>
           </lightning-map>
       </div>
   </template>

   <!-- While request is processing, show spinner -->
   <div class="slds-m-around_large">
       <template lwc:if={requestInProgress}>
           <div class="slds-is-relative">
               <lightning-spinner
                   alternative-text="Getting location...">
               </lightning-spinner>
           </div>
       </template>
   </div>

   <!-- Static help text -->
```

```
    <div class="slds-text-align_center slds-text-color_weak slds-m-vertical_large">
        Click <strong>Get Current Location</strong> to see where you are.
    </div>

    <!-- The get-current-location button;
         Disabled if LocationService isn't available -->
    <div class="slds-align_absolute-center slds-m-vertical_large">
        <lightning-button
            variant="brand"
            disabled={locationButtonDisabled}
            icon-name="utility:target"
            label="Get Current Location"
          title="Use your device's GPS and other location sensors to determine your current
 location"
            onclick={handleGetCurrentLocationClick}>
        </lightning-button>
    </div>
</template>
```

Once the current location is determined, we use the `lightning-map` base component to display it. Each phase of the location request lifecycle writes a console message.

```
// locationServiceExample.js
import { LightningElement } from 'lwc';
import { ShowToastEvent } from 'lightning/platformShowToastEvent';
import { getLocationService } from 'lightning/mobileCapabilities';

export default class LocationServiceExample extends LightningElement {

    // Internal component state
    myLocationService;
    currentLocation;
    locationButtonDisabled = false;
    requestInProgress = false;

    // When component is initialized, detect whether to enable Location button
    connectedCallback() {
        this.myLocationService = getLocationService();
        if (this.myLocationService == null || !this.myLocationService.isAvailable()) {
            this.locationButtonDisabled = true;
        }
    }

    handleGetCurrentLocationClick(event) {
        // Reset current location
        this.currentLocation = null;

        if(this.myLocationService != null && this.myLocationService.isAvailable()) {

            // Configure options for location request
            const locationOptions = {
                enableHighAccuracy: true
            }

            // Show an "indeterminate progress" spinner before we start the request
```

```
            this.requestInProgress = true;

            // Make the request
            // Uses anonymous function to handle results or errors
            this.myLocationService
                .getCurrentPosition(locationOptions)
                .then((result)  => {
                    this.currentLocation = result;

                    // result is a Location object
                    console.log(JSON.stringify(result));

                    this.dispatchEvent(
                        new ShowToastEvent({
                            title: 'Location Detected',
                            message: 'Location determined successfully.',
                            variant: 'success'
                        })
                    );
                })
                .catch((error) => {
                    // Handle errors here
                    console.error(error);

                    // Inform the user we ran into something unexpected
                    this.dispatchEvent(
                        new ShowToastEvent({
                            title: 'LocationService Error',
                            message:
                                'There was a problem locating you: ' +
                                JSON.stringify(error) +
                                ' Please try again.',
                            variant: 'error',
                            mode: 'sticky'
                        })
                    );
                })
                .finally(() => {
                    console.log('#finally');
                    // Remove the spinner
                    this.requestInProgress = false;
                });
        } else {
            // LocationService is not available
            // Not running on hardware with GPS, or some other context issue
            console.log('Get Location button should be disabled and unclickable. ');
            console.log('Somehow it got clicked: ');
            console.log(event);

            // Let user know they need to use a mobile phone with a GPS
            this.dispatchEvent(
                new ShowToastEvent({
                    title: 'LocationService Is Not Available',
                    message: 'Try again from the Salesforce app on a mobile device.',
```

```
                    variant: 'error'
                })
            );
        }
    }

    // Format LocationService result Location object as a simple string
    get currentLocationAsString() {
        return `Lat: ${this.currentLocation.coords.latitude}, Long:
${this.currentLocation.coords.longitude}`;
    }

    // Format Location object for use with lightning-map base component
    get currentLocationAsMarker() {
        return [{
            location: {
                Latitude: this.currentLocation.coords.latitude,
                Longitude: this.currentLocation.coords.longitude
            },
            title: 'My Location'
        }]
    }
}
```

SEE ALSO:

Use the LocationService API

Compatibility and Requirements

LocationService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. LocationService **requires** access to device hardware and device platform APIs. This access is only available when LocationService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

LocationService is available in Lightning apps distributed using:

- Salesforce Mobile app
- Mobile Publisher for Salesforce App
- Mobile Publisher for Experience Cloud
- Field Service Mobile app

LocationService is fully functional when used in a Lightning app or Lightning site run from one of these Salesforce apps on a compatible iOS or Android mobile device. See Requirements for the Salesforce Mobile App, Requirements for Mobile Publisher for Salesforce App, Field Service Mobile App Requirements, or the requirements page for your target mobile app, for specific device and operating system requirements.

LocationService is **not** fully functional when running on other devices, such as a desktop, or when running in a standard web browser, even on a mobile device. It **requires** one of the mobile apps listed above. The LocationService API is accessible in Lightning Experience on all devices, so your code won't fail due to missing functions. However, when your app runs in a browser—desktop or mobile—it can use only LocationService constants and utility functions. Attempting to use location-specific features will fail.

LocationService only provides location data, in the form of geo-coordinates and some metadata. What you do with the location data is up to your component to parse, compare, and place in context—for example, to calculate distance, display on a map, and so on.

Considerations and Limitations

Keep the following in mind when developing features that use the LocationService API.

Device Limitations

LocationService doesn't implement location calculation itself. Instead, it makes available certain location features of the underlying platform (Android or iOS) and hardware (phone or other mobile device). While the features provided by LocationService are the same across both platforms, they are subject to some platform-specific quirks and minor differences.

- LocationService requires the use of the mobile device location detection hardware. The user must grant your app access to the device's location. The exact user experience is governed by the platform. The request happens automatically on first use, and is managed by the device itself, but you should plan for it when designing the user experience of your app.
- In Android 11 or later, if the user taps "Deny" for permission to access the mobile device location more than once during the app's lifetime of installation on a device, the user won't see the system permissions dialog again. Tapping Deny multiple times implicitly chooses the "don't ask again" option.

  In previous versions of Android, users would see the system permissions dialog each time the app requested permission unless the user had previously selected "don't ask again". This change in Android 11 discourages repeated requests for permissions that users have chosen to deny.

  If the user has denied permission to access the mobile device location and needs to change their permissions to allow access, they can do so in their device's settings.

- Location tracking is subject to significant privacy, processing, and power use restrictions, imposed by the underlying platform (Android or iOS). See important details in the LocationService API reference documentation.
- Many factors affect the accuracy of location detection, the speed of determining the current location, and how much impact location tracking has on battery life.
- Mobile devices vary in the quality of the location detection they provide.
- If a mobile device can't determine its location, neither can LocationService. The quality of the device's positioning hardware, tall or dense buildings, indoor use, and other external factors can reduce the accuracy of location determination.
- Location tracking on high accuracy can **significantly** increase power use, which affects how quickly a mobile device drains its battery.

If you're having trouble getting LocationService to provide an accurate location, try the following:

- First, verify that the device is able to determine its location. If the standard Maps app can't get an accurate location, neither can LocationService.
- Next, verify that the user has granted location access to the mobile app where your component is running.
- Finally, double-check the configuration of LocationService in your code. In particular, verify that you've set the accuracy level as needed for your component's features.

Development Considerations

LocationService requires access to positioning hardware, such as GPS, Wi-Fi, cellular, and other location-detection hardware of your mobile device. To test location services during development, use actual, physical devices when possible.

- The Android emulator and iOS simulator can each be configured to provide simulated location details.
- Neither virtual device accurately simulates location detection in the real world, especially for environments where location detection is challenging.

You can certainly develop the user experience for your component on a desktop or laptop development system. But be sure to test location-based functionality on the physical devices on which you plan to deploy your Lightning app.

LocationService Considerations

Be aware of the following considerations when using LocationService in your Lightning app.

- LocationService is built on top of mobile operating system and device features. LocationService's location capabilities therefore depend on Android or iOS features, which are subject to change beyond our control. When mobile operating system features change, the behavior of LocationService can change without notice.
- LocationService doesn't provide map data. To display the current location on a map, you need to add extra functionality, for example, to retrieve map tile graphics.
- LocationService detects a device's current location only. While you can receive updates when the current location changes, updates provide only the new location. If needed, perform distance, proximity, or geofencing calculations yourself, in your own JavaScript.

  Important: While it's possible and not too difficult, a manually implemented geofencing feature can result in **severe** battery drain on devices that use it.

Interact with NFC Tags on a Mobile Device

A Lightning web component can use a device's native NFC functionality to read, erase, and write to NFC tags. When an NFC operation is successful, the text data extracted from the NFC tag or a simple success message is returned to the Lightning web component that invoked it.

NFC operations are performed locally on the mobile device, and don't need a network connection. However, NFCService requires access to platform-specific APIs that are available only within compatible Salesforce mobile apps.

Important: NFCService does not and cannot function when running in a web browser, whether on a desktop or mobile device.

IN THIS SECTION:

NFCService User Experience

Your component can deliver any user experience you desire, but there's a common flow for any component that interacts with NFCs.

Use the NFCService API

To develop a Lightning web component capable of interacting with NFCs, use the NFCService API.

NFCService Example

Here's a basic example of a Lightning web component that uses NFCService to parse text data from an image.

Compatibility and Requirements

NFCService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when NFCService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

Considerations and Limitations

Consider these guidelines and limitations when developing features that use the NFCService API.

NFCService User Experience

Your component can deliver any user experience you desire, but there's a common flow for any component that interacts with NFCs.

- User initiates an NFC operation (read, erase, or write).
- OS prompts the user to hold their device near the NFC tag to be interacted with.
- OS provides a success message when the operation completed successfully, or an error message if something went wrong.

Use the NFCService API

To develop a Lightning web component capable of interacting with NFCs, use the NFCService API.

1. Import NFCService into your component definition to make the NFCService API functions available to your code.
2. Test to make sure NFCService is available before you call NFC-related functions.
3. Use the feature functions to perform NFC-related operations.

For complete API reference documentation of the NFCService API, see NFCService API in the Lightning Web Components Developer Guide.

Add NFCService to a Lightning Web Component

In your component's JavaScript file, import NFCService using the standard JavaScript `import` statement. Specifically, import the `getNFCService()` factory function from the `lightning/mobileCapabilities` module, like so:

```
import { getNFCService } from 'lightning/mobileCapabilities';
```

After it's imported into your component, use the factory function to get an instance of NFCService. With your NFCService instance, use the utility functions and constants to verify availability. Then use NFC-related functions to perform the associated functionality.

Test NFCService Availability

NFCService depends on physical device hardware and platform features. A component that uses NFCService renders without errors on a desktop computer or in a mobile browser, but NFC scanning functions fail. To avoid these errors, test if NFCService functionality is available before you use it.

```
handleBeginNFCEventClick(event) {
    const myNFCService = getNFCService();
    if(myNFCService.isAvailable()) {
        // Perform NFC-related operations
    }
    else {
        // NFCService not available, or consuming app hasn't implemented it
        // Not running on hardware with NFC capabilities
        // Handle with message, error, beep, and so on
    }
}
```

Read an NFC Tag

Perform NFC read operations with `read()`. First, call the `read()` function, optionally passing in an `NFCServiceOptions` object as a parameter. Then, handle the result of the operation in any manner you wish.

For example:

```
myNFCService
  .read(options)
  .then((result) => {
    // Do something with the result of the read operation
    console.log(result);
    this.readResult = result.value;
  })
```

```
  .catch((error) => {
    // Handle cancellation and read errors here
    console.error(error);
  });
```

Erase an NFC Tag

Perform NFC erase operations with `erase()`. First, call the `erase()` function, optionally passing in an `NFCServiceOptions` object as a parameter. Then, handle the result of the operation in any manner you wish.

For example:

```
myNFCService
  .erase(options)
  .then((successMessage) => {
    // Receive a success message following a successful erase operation
    console.log(successMessage);
  })
  .catch((error) => {
    // Handle cancellation and erase errors here
    console.error(error);
  });
```

Note: The `erase()` function can only be performed on writable NFC tags. Attempting this function on a read-only NFC tag results in an error.

Write to an NFC Tag

Perform NFC write operations with `write()`. First, call the `write()` function, optionally passing in an `NFCServiceOptions` object as a parameter. Then, handle the result of the operation in any manner you wish.

For example:

```
myNFCService
  .write(options)
  .then((successMessage) => {
    // Receive a success message following a successful write operation
    console.log(successMessage);
  })
  .catch((error) => {
    // Handle cancellation and write errors here
    console.error(error);
  });
```

Note: The `write()` function can only be performed on writable NFC tags. Attempting this function on a read-only NFC tag will result in an error.

NFCService Example

Here's a basic example of a Lightning web component that uses NFCService to parse text data from an image.

The component's HTML template is minimal, with a display view that includes three buttons, one each for read, erase, and write operations.

```
<template>
  <lightning-card title="NFC Service Demo" icon-name="custom:phone_portrait">
    <div class="slds-var-m-around_medium">
      Choose an action to perform on an NFC tag:<br><br>

      <lightning-button
        variant="brand"
        label="Read"
        title="Read the content of an NFC tag"
        onclick={handleReadClick}>
      </lightning-button>
      <lightning-button
        variant="brand"
        label="Erase"
        title="Erase the content of an NFC tag"
        onclick={handleEraseClick}
        class="slds-var-m-left_x-small">
      </lightning-button>
      <lightning-button
        variant="brand"
        label="Write"
        title="Write sample content to an NFC tag"
        onclick={handleWriteClick}
        class="slds-var-m-left_x-small">
      </lightning-button>
    </div>
    <div class="slds-var-m-around_medium">
      <lightning-formatted-text value={status}></lightning-formatted-text>
    </div>
  </lightning-card>
</template>
```

This example uses NFCService to select the NFC operation to be performed, performs the operation, and displays a success message when completed successfully. An error message is returned when there's an error.

```
import { LightningElement } from 'lwc';
import { getNfcService } from 'lightning/mobileCapabilities';

export default class NimbusPluginNfcService extends LightningElement {
    status;
    nfcService;

    connectedCallback() {
      this.nfcService = getNfcService();
    }

    handleReadClick() {
      if(this.nfcService.isAvailable()) {
        const options = {
          "instructionText": "Hold your phone near the tag to read.",
          "successText": "Tag read successfully!"
        };
        this.nfcService.read(options)
```

```
          .then((result) => {
            // Do something with the result
            this.status = JSON.stringify(result, undefined, 2);
          })
          .catch((error) => {
            // Handle errors
           this.status = 'Error code: ' + error.code + '\nError message: ' + error.message;

          });
      } else {
        // service not available
        this.status = 'Problem initiating NFC service. Are you using a mobile device?';
      }
    }

    handleEraseClick() {
      if(this.nfcService.isAvailable()) {
        const options = {
          "instructionText": "Hold your phone near the tag to erase.",
          "successText": "Tag erased successfully!"
        };
        this.nfcService.erase(options)
          .then(() => {
            this.status = "Tag erased successfully!";
          })
          .catch((error) => {
            // Handle errors
           this.status = 'Error code: ' + error.code + '\nError message: ' + error.message;

          });
      } else {
        // service not available
        this.status = 'Problem initiating NFC service. Are you using a mobile device?';
      }
    }

    async handleWriteClick() {
      if(this.nfcService.isAvailable()) {
        const options = {
          "instructionText": "Hold your phone near the tag to write.",
          "successText": "Tag written successfully!"
        };
        const payload = await this.createWritePayload();
        this.nfcService.write(payload, options)
          .then(() => {
            this.status = "Tag written successfully!";
          })
          .catch((error) => {
            // Handle errors
           this.status = 'Error code: ' + error.code + '\nError message: ' + error.message;

          });
      } else {
        // service not available
```

```
        this.status = 'Problem initiating NFC service. Are you using a mobile device?';
      }
    }

    async createWritePayload() {
      // Here we demonstrate how you can write several records to an NFC tag.
      // Consider the scenario where you want to write the content of a business card
      // to an NFC tag. The content can be broken down into a number of text and uri
records.
     const nameRecord = await this.nfcService.createTextRecord({text: "John Smith", langId:
 "en"});
      const phone1Record = await this.nfcService.createTextRecord({text: "(123) 456-7890
Office", langId: "en"});
      const phone2Record = await this.nfcService.createTextRecord({text: "(321) 654-0987
Direct", langId: "en"});
      const emailRecord = await
this.nfcService.createUriRecord("mailto:john.smith@email.com");
     const addressRecord = await this.nfcService.createTextRecord({text: "584 South Paris
 Hill Ave., Lancaster, CA 93535", langId: "en"});
      const websiteRecord = await
this.nfcService.createUriRecord("https://www.mycompany.com");
      return [nameRecord, phone1Record, phone2Record, emailRecord, addressRecord,
websiteRecord];
    }
}
```

Compatibility and Requirements

NFCService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when NFCService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

NFCService is available in Lightning apps distributed using:

- Salesforce Mobile app
- Salesforce Field Service app (Android only)

NFCService is fully functional when used in a Lightning app or Lightning site that's run from a compatible Salesforce mobile app on a compatible iOS or Android mobile device. See Requirements for the Salesforce Mobile App, or the requirements page for your target mobile app for specific device and operating system requirements.

NFCService is **not** fully available when running on other devices, such as a desktop, or when running in a web browser, even on a mobile device. It **requires** one of the apps listed above. The NFCService API is accessible in Lightning Experience on all devices, so your code won't fail due to missing functions. However, when your app runs in a browser—desktop or mobile—it can use only NFCService constants and utility functions. Attempting any NFC-related operation will fail.

Considerations and Limitations

Consider these guidelines and limitations when developing features that use the NFCService API.

Device Limitations

- In Android 11 or later, if the user denies permission to access NFC functionality more than one time during the app's lifetime of installation on a device, the user won't see the system permissions dialog again. Tapping Deny multiple times implicitly chooses the "don't ask again" option.
- In previous versions of Android, users see the system permissions dialog each time the app requested permission unless the user had previously selected "don't ask again". This change in Android 11 discourages repeated requests for permissions that users have chosen to deny.
- If the user has denied permission to access NFC functionality and needs to change their permissions to allow access, they can do so in their device's settings.

Development Considerations

- NFCService requires access to NFC hardware. To test NFC interactions during development, use actual, physical devices.

Warning: NFCService allows for an action that, if used irresponsibly or incorrectly, can lead to irreversible consequences for your users. This dangerous action is the capability to erase NFC tags from a mobile device.

As with all mobile capabilities, implementation of NFCService's functionality, dangerous actions included, is at your discretion. Use caution when using dangerous actions in your component.

NFCService Considerations

- NFCService is built on top of mobile operating system and device features. NFCService's capabilities therefore depend on Android or iOS features, which are subject to change beyond our control. When mobile operating system features change, the behavior of NFCService can change without notice.

Accept On-Site Payments with Tap-to-Pay

A Lightning web component can use a mobile device to let your customers use the Tap-to-Pay capability of the Payments plug-in to pay mobile workers directly. The Field Service mobile app then integrates with Pay Now to connect the Lightning web component to a secure payment system that processes the interaction.

The PaymentsService plugin allows Field Service mobile workers to collect payments from their customers using Tap to Pay. This service integrates with Salesforce Payments and Stripe as a payment provider. Tap to Pay is supported on iOS and Android devices with Stripe as the payment provider.

Important: PaymentsService does not and cannot function when running in a web browser, whether on a desktop or mobile device.

IN THIS SECTION:

PaymentsService User Experience

Your component can deliver any user experience you want, but you must follow a common flow for any component that calls for a Payment Service.

Use the PaymentsService API

To develop an LWC with the Payments Service plug-in features, use the Payments Plugin API as your method for accessing a device's native Tap to Pay functionality.

PaymentsService Example

Here's a basic example of a Lightning web component minimal HTML template that includes a button that initiates collecting payment.

Compatibility and Requirements

PaymentsService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when PaymentsService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

Considerations and Limitations

Consider these guidelines and limitations when developing features that use the PaymentsService API.

SEE ALSO:

Lightning Web Components Developer Guide: PaymentsService API

PaymentsService User Experience

Your component can deliver any user experience you want, but you must follow a common flow for any component that calls for a Payment Service.

The user performs an action that triggers a collect payment flow. The operating system provides messages for the user to tap their credit card in the right direction. If the collect payment method isn't successful, the plug-in provides an error message. If it's successful, a PaymentIntent object is returned to the user, so they can perform any other follow-up actions on the Stripe dashboard, such as refund or cancel the payment, if needed.

Use the PaymentsService API

To develop an LWC with the Payments Service plug-in features, use the Payments Plugin API as your method for accessing a device's native Tap to Pay functionality.

1. Import PaymentsService to make the PaymentsService API functions available to your code.
2. Test to make sure PaymentsService is available before you call payment functions.
3. Use the payment functions to start collecting payments.

Add PaymentsService to a Lightning Web Component

In your component's JavaScript file, import PaymentsService using the standard JavaScript `import` statement. Specifically, import the `getPaymenntsService()` factory function from the `lightning/mobileCapabilities` module, like so:

```
import { getPaymentsService } from 'lightning/mobileCapabilities';
```

After it's imported into your component, use the factory function to get an instance of PaymentsService. With your PaymentsService instance, use the utility functions and constants to verify availability. Then use the feature functions to perform the associated functionality.

Test PaymentsService Availability

PaymentsService depends on physical device hardware and platform features. A component that uses PaymentsService renders without errors on a desktop computer or in a mobile browser, but PaymentsService functions fail. To avoid these errors, test if PaymentsService functionality is available before you use it.

```
handleCheckCollectPaymentServiceClick(event) {
    const myPaymentsService = getPaymentsService();
    if(myPaymentsService.isAvailable()) {
```

```
  // Perform next operations
  } else {
 // Payments Service isn't available, or consuming app hasn't implemented it
 // Not running on hardware with TTP functionality, etc.
 // Handle with message, error, beep, and so on
  }
}
```

Start PaymentsService to Collect Payments

The PaymentsService exposes two API endpoints. The first one is `getSupportedPaymentMethods` that returns a list of available payment methods for the current device running. The second API method is called `collectPayment`.

Start by checking which payment methods are available, and then call the collect payment method to collect the payment using one of the available methods.

```
handleGetSupportedMethodsClicked(event) {
 if (this.myPaymentsService != null && this.myPaymentsService.isAvailable()) {
 let supportedMethodsOptions = {
  countryIsoCode: "USD",
  merchantAccountId: "8zbxxxxxxxxxxx"
 }
this.myPaymentsService.getSupportedPaymentMethods(supportedMethodsOptions).then((supportedMethodsResult)
 => {
if (supportedMethods.contains("TAP_TO_PAY")) {
  let collectPaymentOptions = {
  amount: "350.50",
  paymentMethod: "TAP_TO_PAY"
  currencyIsoCode: "USD",
  merchantAccountId: "8zbxxxxxxxxxxx",
  merchantName: "My Service",
  payerAccountId: "001xxxxxxxxxx",
  sourceObjectIds: ["xxxxxxxxxxxx", "xxxxxxxxxxx"]
 }
this.myPaymentsService.collectPayment(collectPaymentOptions).then((collectPaymentResult)
=> {
let paymentStatus = collectPaymentResult.status;
 // handle status. "success" status will reflect a successful payment collect
}).catch((collectPaymentError) => {
if (collectPaymentError.code !== 'USER_DISMISSED') {
// handle error case
}
})


}
}).catch((error) => {
 console.log(error);
});
}
}
```

Considerations:

- Calling `getSupportedPaymentMethods` or `collectPayment` requires passing in an options object.

- The options object has some required fields and missing those fields results in the Payments plugin not working properly. See the PaymentsService API for the required fields.

SEE ALSO:

Lightning Web Components Developer Guide: PaymentsService API

PaymentsService Example

PaymentsService Example

Here's a basic example of a Lightning web component minimal HTML template that includes a button that initiates collecting payment.

```
<template>
    <lightning-card title="Payment Processing">
        <div class="slds-m-around_medium">
            <lightning-button
                label="Validate Supported Methods"
                onclick={handleValidateMethods}
                variant="neutral"
                class="slds-m-right_x-small">
            </lightning-button>
            <lightning-button
                label="Collect Payment"
                onclick={handleCollectPayment}
                variant="brand">
            </lightning-button>
        </div>
<div class="slds-var-m-around_medium">
      <div if:true={spinnerEnabled} class="spinnerHolder">
       <lightning-spinner alternative-text="Processing" size="large"></lightning-spinner>

      </div>
      <lightning-formatted-rich-text
value={paymentsServiceResponse}></lightning-formatted-rich-text>
    </div>
    </lightning-card>
</template>
```

This example uses PaymentsService to let the user collect a payment.

```
import { LightningElement, track } from 'lwc';
import { ShowToastEvent } from 'lightning/platformShowToastEvent';
import { getPaymentsService } from 'lightning/mobileCapabilities';


export default class PaymentsService extends LightningElement {
    myPaymentsService;
    paymentServiceUnavailable = false;
    tapToPayUnAvailable = true;

    paymentsServiceResponse = '';
    spinnerEnabled = false;
    amountValue = '150.00';
    currencyCodeValue = 'USD';
```

```
    paymentMethodValue = 'TAP_TO_PAY';
    merchantNameValue = 'Play Board';
    merchantAccountIdValue = '8zbXXXXXXXXXXXX';
    payerAccountIdValue = '001XXXXXXXXXXXX';
    sourceObjectIdValue = 'XXXXXXXXXXXXXXXX';


    showSpinner() { this.spinnerEnabled = true; }
    hideSpinner() { this.spinnerEnabled = false; }


    // When the component is initialized, determine whether to enable the buttons
    connectedCallback() {
        this.myPaymentsService = getPaymentsService();
        if (!this.myPaymentsService?.isAvailable()) {
            this.paymentServiceUnavailable = true;
            this.paymentsServiceResponse = 'Payments Service is unavailable.';
        }
    }


    processResult(result) {
  var confirmationId = JSON.stringify(result.guid, undefined, 2);
  var confirmationStatus = JSON.stringify(result.status, undefined, 2);
        this.paymentsServiceResponse = "Confirmation Status: " + confirmationStatus + ",
Confirmation ID: " + confirmationId;
    }


    processError(error) {
        // The user canceled the operation
        if (error.code === 'USER_DISMISSED') {
            this.dispatchEvent(
                new ShowToastEvent({
                    title: 'Operation Canceled',
                    message: 'Operation canceled by the user.',
                    mode: 'sticky'
                })
            );
        }
        else {
        // There was some other kind of error so inform the user we ran into something
unexpected


            this.dispatchEvent(
                new ShowToastEvent({
                    title: 'Payments Service Error',
                    message: 'Message: ${error.message}\nCode: ${error.code}',
                    variant: 'error',
                    mode: 'sticky'
                })
            );
        }
```

```
    }


    handleBeginSupportedMethodsClick() {
        let options = {
            'countryIsoCode': this.currencyCodeValue,
            'merchantAccountId': this.merchantAccountIdValue
        }


        this.paymentsServiceResponse = '';
        this.showSpinner();


        this.myPaymentsService.getSupportedPaymentMethods(options)
        .then((result) => {
   if (result.contains("TAP_TO_PAY")) {
                this.tapToPayUnAvailable = false;
}

        })
        .catch((error) => {
            this.tapToPayUnAvailable = true;
        })
        .finally(() => this.hideSpinner());
    }


    handleBeginCollectPaymentClick() {

        let options = {
            'amount': Number(this.amountValue),
            'currencyIsoCode': this.currencyCodeValue,
            'paymentMethod': this.paymentMethodValue,
            'merchantName': this.merchantNameValue,
            'merchantAccountId': this.merchantAccountIdValue,
            'payerAccountId': this.payerAccountIdValue,
            'sourceObjectIds': [this.sourceObjectIdValue]
        }


        this.paymentsServiceResponse = '';
        this.showSpinner();

        this.myPaymentsService.collectPayment(options)
        .then((result) => this.processResult(result))
        .catch((error) => this.processError(error))
        .finally(() => this.hideSpinner());
    }
}
```

SEE ALSO:

Use the PaymentsService API

Compatibility and Requirements

PaymentsService is a JavaScript module that provides an API to Lightning web components to make mobile hardware and platform (operating system) features available in JavaScript. It **requires** access to device hardware and device platform APIs. This access is only available when PaymentsService runs within a compatible Salesforce mobile app. **It does not and cannot function when running in a standard web browser, whether the browser runs on a desktop or mobile device.**

PaymentsService is available in Lightning apps distributed using:

- Salesforce Field Service app

PaymentsService is fully functional when used in a Lightning app or Lightning site that's run from a compatible Salesforce mobile app on a compatible iOS or Android mobile device. See Field Service Mobile App Requirements, or the requirements page for your target mobile app for specific device and operating system requirements.

PaymentsService is **not** fully available when running on other devices, such as a desktop, or when running in a web browser, even on a mobile device. It **requires** one of the listed apps. The PaymentsService API is accessible in Lightning Experience on all devices, so your code won't fail due to missing functions. However, when your app runs in a browser—desktop or mobile—it can use only PaymentsService constants and utility functions. Attempting any payment-related operation results in failure.

Considerations and Limitations

Consider these guidelines and limitations when developing features that use the PaymentsService API.

PaymentsService Considerations

Be aware of the following considerations when using PaymentsService in your Lightning app.

- PaymentsService is built on top of mobile operating system and device features. PaymentsService's capabilities therefore depend on Android or iOS features, which are subject to change beyond our control. When mobile operating system features change, the behavior of PaymentsService can change without notice.
- Calling `getSupportedPaymentMethods` or `collectPayment` requires passing in an options object.
- The options object has some required fields and missing those fields will result in the Payments plugin not working properly. See the PaymentsService API for the required fields.

CHAPTER 4 Offline Considerations and Limitations

In this chapter ...

- General Considerations
- Considerations for Field Service Mobile App
- Base Components Support
- Modules Support
- Wire Adapters Support
- Entity Support
- Metadata and Custom Metadata Types Support

LWC Offline is designed to let you build great apps that can function without a network connection, but it's *not* the full Salesforce service. Lightning web components have a number of limitations when used offline, including missing capabilities, reduced performance, and software defects (bugs). Keep these limitations in mind as you design and develop your offline customizations.

General Considerations

Consider these general details when planning your Lightning web components development efforts.

- The Lightning Web Components framework includes a wide range of built-in components, a number of modules that enable features, and a range of wire adapters for data access. The implementation available in LWC Offline supports a subset of these features. See additional details elsewhere in this guide.
- You can only use Lightning web components that are used as global or object-specific quick actions.
- Object-specific quick actions can only be added to record detail pages.
- Your components must be Lightning **web** components.
  - You can't use Aura-based Lightning components, despite the similar name.
  - You can't use Visualforce at all with Lightning web components.

- Headless quick actions aren't supported at this time.
- Deep links to quick actions are supported. Deep links to **global** quick actions aren't supported at this time.
- To work offline, your custom components must be statically analyzable by the Salesforce service so that they can be preloaded before going offline. See Offline Environment Details for details.
- Calling `alert()` from JavaScript in a Lightning web component is unsupported. It's also an anti-pattern in Lightning web components. For debugging and logging, use `console.log()` and `console.error()`. Better yet, use Chrome DevTools or Safari Web Inspector.
- For user-facing messages, the correct pattern is to use a toast message. However, support for the `lightning/platformShowToastEvent` module is incomplete. Use `LightningAlert` instead.
- Lightning web components perform minimal validations while offline. It's possible for a record to be changed, or a new record created, which passes local validation while offline. However, it's possible for this record to subsequently fail server-side validation when the draft record is uploaded. Record drafts that fail server-side validation block the offline queue and prevent record changes from uploading. Manually clear the invalid record to unblock the queue.
- Depending on context, it can be unclear which field of a draft record has failed server-side validation.
- Some Lightning web components don't render properly due to incorrect form factor detection in certain circumstances. See the known issues in Modules Support for details.
- A maximum of 50 records are fetched for each related list. The list size indicator — for example, "(50+)" — reflects the number of records downloaded to the app, not the number of records that exist. This limit will be customizable in a future release.

Considerations for Field Service Mobile App

The following considerations apply to LWC Offline when you run your components in the Field Service Mobile app.

These considerations apply to LWCs *only* when run in the Field Service mobile app.

- Global quick actions are available on all pages that have the Actions menu. They do **not** receive the record ID of the current record when invoked from a record detail page.
- A Community license user who opens a Service Appointment can experience a missing record error. This is a known issue with this specific user type.
- Task objects added to a briefcase aren't primed. If a briefcase contains a Task object and priming is attempted, an error message is displayed. However, other objects in the briefcase are primed. To resolve the error, remove Task objects from the briefcase.
- There's a conflict between Appointment Assistance and LWC Offline that can result in missing URLs. See this known issue for details.

- (iOS only) Changes to Lightning web components are loaded into the app only when the app is fully quit and then relaunched (a "cold start").

Base Components Support

Base components are described in the Lightning Web Components Reference. All components are being reviewed for correct behavior in LWC Offline-enabled mobile apps.

The following table provides details for base components where support is incomplete or not available. Base components that are generally available but not listed here are supported.

- ⚠ — **Limited Support**. Can be used, but has known (and possibly unknown) issues.
- ⓧ — **Not Supported**. Doesn't work, or shouldn't be used.

Component	Status	Comments
`lightning-combobox`	⚠	Supported for use, but not mobile friendly.
`lightning-datatable`	ⓧ	This complex component was never designed for mobile use, and its behavior in mobile contexts is subject to change without notice. See this post in the Trailblazer community for sample code for a simplified, mobile-friendly data table.
`lightning-file-upload`	ⓧ	
`lightning-formatted-address`	⚠	The static map feature only works in online mode. If a user is offline, an error message is displayed instead.
`lightning-formatted-date-time`	⚠	Supported for en-US locale only.
`lightning-formatted-rich-text`	⚠	URLs don't open within a navigation view; external URLs work, but links to records don't. Additionally, you can't open any URL with a `target="_blank"` attribute.
`lightning-formatted-time`	⚠	Supported for en-US locale only.
`lightning-formatted-url`	⚠	URLs don't open within a navigation view; external URLs work, but links to records don't. Additionally, you can't open any URL with a `target="_blank"` attribute.
`lightning-input`	⚠	Supported for en-US locale only.
`lightning-input-address`	⚠	The address lookup feature only works in online mode. If a user is offline, an error message is displayed instead.
`lightning-input-rich-text`	⚠	Associated components, `lightning-rich-text-toolbar-button` and `lightning-rich-text-toolbar-button-group`, aren't supported. Image uploading within a rich text field isn't supported.

Component	Status	Comments
lightning-map	⚠	This component requires a connection to a mapping service, and can't work while offline. If users access this component while offline, an error message is displayed instead.
lightning-pill	⚠	URLs don't open within a navigation view; external URLs work, but links to records don't.
lightning-pill-container	⚠	URLs don't open within a navigation view; external URLs work, but links to records don't.
lightning-record-form	⚠	lightning-record-form doesn't work with draft records, that is, records created while offline. See Considerations for getRecordUi for an explanation of the limitations. Lookup fields are read-only. Getters and setters aren't supported in offline priming. In your custom components, make sure attributes passed to lightning-record-form are either hardcoded or have a simple API.
lightning-record-edit-form	⚠	lightning-record-edit-form doesn't work with draft records, that is, records created while offline. See Considerations for getRecordUi for an explanation of the limitations. Lookup fields are read-only. Getters and setters aren't supported in offline priming. In your custom components, make sure attributes passed to lightning-record-edit-form are either hardcoded or have a simple API. If the input field isn't on the page layout, you must pass the field in using optional-fields.
lightning-record-view-form	⚠	lightning-record-view-form doesn't work with draft records, that is, records created while offline. See Considerations for getRecordUi for an explanation of the limitations. Getters and setters aren't supported in offline priming. In your custom components, make sure attributes passed to lightning-record-view-form are either hardcoded or have a simple API. If the input field isn't on the page layout, you must pass the field in using optional-fields.
lightning-tab	⚠	Supported for use, but not mobile friendly.
lightning-tabset	⚠	Supported for use, but not mobile friendly.

Component	Status	Comments
lightning-tree-grid	⊗	

Additional Component Considerations

Several localization and globalization issues affect a number of components. These issues generally affect formatting of dates, times, currencies, and numbers. They're mostly irritating, but only cosmetic. However, a few can cause errors, most often due to incorrect processing of numbers with too many digits (15 or more digits) in them. If you encounter these, try rounding the number manually in JavaScript, to no more than 14 digits of precision, before passing it to a base component.

None of the `lightningsnapin-*` components are supported.

SEE ALSO:

Component Reference

Modules Support

Lightning web component modules in the `lightning` namespace are described in the Component Reference in the Lightning Web Components Developer Guide. Modules scoped with `@salesforce` are described in `@salesforce` Modules in the Lightning Web Components Developer Guide. All modules are being reviewed for correct behavior in LWC Offline-enabled mobile apps.

The following table presents current findings.

- ✓ — **Supported**. Expected to behave as documented.
- ⚠ — **Limited Support**. Can be used, but has known (and possibly unknown) issues.
- ⊗ — **Not Supported**. Doesn't work, or shouldn't be used.

Module	Status	Comments
`lightning` **Namespace Modules** These modules contain resources that don't change and are universal to all orgs.		
`lightning/alert` Create an alert modal within your component.	✓	
`lightning/confirm` Create a confirm modal within your component.	✓	
`lightning/empApi` Provides methods for subscribing to a streaming channel and listening to event messages.	⊗	Not supported in mobile apps.
`lightning/flowSupport`	⊗	

Module	Status	Comments
Provides events to control flow navigation and notify the flow of changes in attribute values.		
`lightning/messageService` Communicates across the DOM between Visualforce pages, Aura components, and Lightning web components.		Lightning Aura components and Visualforce aren't supported in the Field Service mobile app. This isn't expected to change.
`lightning/navigation` Generates a URL or navigates to a page reference.		Available for use, but supported page reference types are limited. See Navigation for details.
`lightning/pageReferenceUtils` Provides utilities for encoding and decoding default field values.		Not supported in mobile apps.
`lightning/platformResourceLoader` Imports a third-party JavaScript or CSS library.		Not supported for offline use.
`lightning/platformShowToastEvent` Displays toasts to provide feedback to a user following an action, such as after a record is created.		You can import this module and fire toast events. However, toast messages aren't handled or displayed. Use `lightning/alert`, `lightning/confirm`, or `lightning/prompt` instead.
`lightning/prompt` Create a prompt modal within your component.		
`lightning/userConsentCookie` Utility functions to incorporate the Cookie Consent mechanism in your components.		Relevant only for Experience Builder pages.
`@salesforce` **Scoped Modules** The shape of these modules can be dynamic, defined by your organization's metadata.		
`@salesforce/apex` Import Apex methods as functions that a component can call either via `@wire` or imperatively.		See Use Apex While Mobile and Offline for usage details.
`@salesforce/apexContinuation` Import methods from Apex continuation classes.		See Use Apex While Mobile and Offline for usage details.
`@salesforce/client/formFactor` Import a name that refers to the form factor of the hardware running the app.		This module "works," but always returns `Small` when used in an LWC Offline-enabled app.

Module	Status	Comments
`@salesforce/community` Import the ID of the current Experience Builder site.		Relevant only for Experience Builder pages.
`@salesforce/contentAssetUrl` Import content asset files.		
`@salesforce/i18n` Import internationalization properties.		
`@salesforce/label` Import labels defined in your Salesforce organization.		
`@salesforce/messageChannel` Import a Lightning message channel that a component can use to communicate via the Lightning Message Service.		Lightning Message Service isn't supported in LWC Offline-enabled mobile apps.
`@salesforce/resourceUrl` Import static resources defined in your Salesforce organization.		
`@salesforce/schema` Import references to Salesforce objects and fields defined in your org.		
`@salesforce/user` Import the current user's ID.		
`@salesforce/userPermission` Import a permission and check whether it's assigned to the current user.		
`@salesforce/customPermission` Import a custom permission and check whether it's assigned to the current user.		

SEE ALSO:

Component Reference

Lightning Web Components Developer Guide: `@salesforce` Modules

Wire Adapters Support

Lightning web component wire adapters and JavaScript functions are described in "`lightning/ui*Api` Wire Adapters and Functions" in the *Lighting Web Components Developer Guide*.

The following wire adapters and functions can be used.

Support Status

- ✅ — **Supported**. Expected to behave as documented.
- ⚠️ — **Limited Support**. Can be used, but has known (and possibly unknown) issues.
- ❌ — **Not Supported**. Doesn't work, and shouldn't be used.

Offline Capability

- **Drafts-Enabled**. Supports creation and modification of records while offline.
- **Offline-Supported**. Supports offline read-only use of primed data while offline, but *not* creation or modification.

Wire Adapter	Status	Offline Capability	Comments
`lightning/uiRecordApi` Read record data and default values. Create, update, delete, and refresh records.			
`createRecord`	✅	Drafts-Enabled	
`createRecordInputFilteredByEditedFields`	❌		
`deleteRecord`	✅	Drafts-Enabled	
`generateRecordInputForCreate`	❌		
`generateRecordInputForUpdate`	❌		
`getFieldValue`	✅		
`getFieldDisplayValue`	✅		
`getRecord`	⚠️	Drafts-Enabled	`getRecord` supports two ways to specify which fields to load: • explicitly, by providing a fields list, or • implicitly, by providing a layout that contains the desired fields. At this time, you must provide a specific list of fields; `getRecord` by layout isn't supported.
`getRecords`	✅	Drafts-Enabled	

Wire Adapter	Status	Offline Capability	Comments
getRecordCreateDefaults		Offline-Supported	
getRecordNotifyChange			
getRecordUi (deprecated)			See Wire Adapter Considerations.
updateRecord		Drafts-Enabled	

Wire Adapter	Status	Offline Capability	Comments
lightning/uiObjectInfoApi Get object metadata, and get picklist values.			
getObjectInfo		Offline-Supported	
getObjectInfos		Offline-Supported	
getPicklistValues		Offline-Supported	
getPicklistValuesByRecordType		Offline-Supported	

Wire Adapter	Status	Offline Capability	Comments
lightning/uiAppsApi **(beta)** Get data and metadata for apps displayed in the Salesforce UI.			
getNavItems (beta)			Not yet supported due to beta status.

Wire Adapter	Status	Offline Capability	Comments
lightning/uiListApi **(deprecated)** Get records and metadata for a list view.			
getListUi (deprecated)			See Wire Adapter Considerations.

Wire Adapter	Status	Offline Capability	Comments
lightning/uiListsApi Get metadata for a list view.			

Wire Adapter	Status	Offline Capability	Comments
`getListInfoByName`		Offline-Supported	Use this adapter instead of `lightning/uiListApi.getListUi`.
`getListInfosByName`			

Wire Adapter	Status	Offline Capability	Comments
`lightning/uiRelatedListApi` Get records, metadata, and record count for a related list.			
`getRelatedListRecords`		Offline-Supported	`getRelatedListRecords` works while offline, but doesn't update to add or remove records that are created or deleted while offline.
`getRelatedListRecordsBatch`			
`getRelatedListInfo`		Offline-Supported	
`getRelatedListInfoBatch`		Offline-Supported	
`getRelatedListsInfo`		Offline-Supported	
`getRelatedListCount`		Offline-Supported	`getRelatedListCount` works while offline, but doesn't update to add or remove records that are created or deleted while offline.

Wire Adapter Considerations

We describe `getRecordUi` and `getListUi` as having Limited Support. Both adapters are deprecated for all customers, and each has additional considerations for offline use. `getRecordUi` in particular has significant limitations. We would prefer to note both of these adapters as Not Supported, but each provides functionality that's not easily replaced today. We recommend you carefully limit your use of these wire adapters.

Note: Forward looking statement: Our goal is to provide supported alternatives to `getRecordUi` and `getListUi`. If you limit your usage of these adapters today, you'll have an easier time migrating later.

- In the case of `getRecordUi`, limit yourself to getting layout metadata details, and use the data-only adapters `getRecord` and `getRecords` for data access.
- In the case of `getListUi`, use the new `getListInfoByName` wire adapter to get list view metadata. If you must use `getListUi` to access list view records, see the following considerations.

Considerations for List Adapters

List- or collection-oriented adapters such as `getListUi`, `getRelatedListRecords`, and `getRelatedListCount` have limited support for offline updates. Specifically, list logic isn't re-evaluated for changes made while you're offline. That is, if you create or modify a record offline and it falls into or out of the list criteria, the record isn't added or removed from the list until you're back online.

The list updates only after the changes sync back to Salesforce. This limitation affects list *membership*, but does not affect the *display* of records that are a part of a list.

Here's an example to make this clear. Let's say you use `getRelatedListRecords` as a source to display a list of records, and the related list criteria limits list membership to accounts whose name begins with "A". While offline, if you update one of those records to change an account name to begin with a "B", from "Apple" to "Banana", that record will still display in the list, with the **updated** account name "Banana". Once you return online, the change syncs to Salesforce, and the list criteria is reevaluated. The Banana account will no longer be a member of the related list, and the wire adapter is updated, triggering a component refresh. The list of records returned by `getRelatedListRecords` won't include the record for the Banana account, and it will disappear from the list displayed in your component's user interface.

Considerations for `getRecordUi`

`getRecordUi` is affected by numerous issues when used while offline.

- Invoking `getRecordUi` on a draft record that was **created** while offline returns an error.
- If you edit a record such that its layout changes—for example, by changing the record type—the results of invoking `getRecordUi` on that record can be inconsistent.
- If you change a relationship field on a record, and the new relationship references a record with a different object type or record type, the results of invoking `getRecordUi` on that record can be inconsistent or result in an error.

In theory, if you're able to limit changes to records while offline to the scalar (non-relationship, non-metadata affecting) fields of that record, then invoking `getRecordUi` on that record should work as documented. In practice this is challenging, and when you miss it results in inconsistent or incorrect behavior that can be hard to troubleshoot. If you must use it, exercise extreme caution.

`getRecordUi` is used in the implementations of the following Lightning base components, causing them to have similar limitations:

- `lightning-record-form`
- `lightning-record-edit-form`
- `lightning-record-view-form`

Handle Errors Defensively

When handling errors returned by wire adapters there's potentially an issue with the "shape" of the error response. In contexts outside LWC Offline, the response returns a single error object. However, when an LWC wire adapter receives an error running in an LWC Offline-enabled mobile app, the response is returned as an **array** of error objects—most often, an array containing just one error object.

To make your components compatible across environments, we recommend a small amount of defensive coding at the start of your error handling. Convert a non-array into an array to ensure that the error shape is consistent:

```
let errors = ...; // errors from wire adapter
if ( ! Array.isArray(errors) ) {
    errors = [ errors ];
}
```

SEE ALSO:

Lightning Web Components Developer Guide: `lightning/ui*Api` Wire Adapters and Functions

Entity Support

LWC Offline uses the UI-API to access entity data. The UI-API supports a long list of standard entities, and all custom entities.

For a complete list of supported entities, see Supported Objects in the *User Interface API Developer Guide*.

Note: The ContentDocument, ContentVersion, and associated entities, used as part of file uploading features, aren't fully supported at this time. This limitation affects all base components that provide file uploading functionality. See Upload Photos from LWCs Using `lightning-input` Base Component in the Trailblazer community for additional details and a partial work-around.

The following entities aren't fully supported by UI-API, but are commonly used in Field Service. Support for them is incomplete.

- Case (not supported in related lists)
- LinkedArticleTask (recurrence isn't supported)
- WorkOrder (not supported in related lists)
- WorkOrderStatus (not supported in related lists)
- WorkOrderLineItem (not supported in related lists)

SEE ALSO:

User Interface API Developer Guide: Supported Objects

User Interface API Developer Guide

Metadata and Custom Metadata Types Support

LWC Offline uses the UI-API to access standard metadata.

Metadata for entities, layouts and other customizations, and Lightning web components is automatically primed or loaded when used, and is cached for offline use. However, the UI-API doesn't support loading custom metadata types. As a consequence, custom metadata isn't primed or cached automatically. This affects features that use custom metadata, such as Flows.

If you must retrieve custom metadata, you can do that using Apex requests. If your Apex request methods are cacheable, the custom metadata you access is available while offline. See Use Apex While Mobile and Offline.

SEE ALSO:

User Interface API Developer Guide

CHAPTER 5 Offline Environment Details

In this chapter ...

- What Happens When Something Isn't Primed (Preloaded)
- Create Components with Offline Analysis In Mind
- Determine Online or Offline Status

The offline environment presents a number of technical challenges for LWC Offline-enabled mobile apps, and for your Lightning web components.

Salesforce mobile apps are heavily optimized for offline use. These optimizations are enabled by our complete understanding of how the built-in features are implemented, including code, data, and metadata relationships. The app contains or loads the information it needs for built-in features to perform well, online and offline. See the following topics in Salesforce Help for application-specific details.

- Salesforce Mobile App Plus: Offline Behaviors
- Field Service Mobile App: Offline Priming in the Field Service Mobile App

Salesforce mobile apps don't, and can't, have this same level of knowledge about your custom features built with Lightning web components. Instead, the app analyzes your custom objects, page layouts, components, and other metadata, and then loads the data and metadata it thinks you need. To improve the quality of this analysis, and thus the performance of your components, you must follow a number of guidelines when developing your Lightning web components.

What Happens When Something Isn't Primed (Preloaded)

Priming for offline use is a "best effort" mechanism. Salesforce mobile apps are resilient in situations where resources are required by a mobile client but weren't primed.

- If the client is online, missing data and metadata resources are loaded when needed. There's a minor performance impact due to the extra network requests.
- If the client is offline, then missing data and metadata can't be retrieved. The Lightning Web Components framework, and the Salesforce mobile app it's running within, handle this situation with either a status display or an error message, depending on what wasn't primed before going offline. Specific behavior is dependent on the component and how it's implemented.

Create Components with Offline Analysis In Mind

To use a feature implemented with Lightning web components while offline, it must be preloaded, or primed, **before** you go offline.

Specifically, an LWC Offline-enabled mobile app must prime:

- The component, and all its dependencies.
- The data to be displayed by the component, and all its dependencies.

The process for determining and resolving component dependencies is complex, and our implementation is continually improving. The fundamental aspect to understand is that this dependency resolution is done without executing or rendering the component. Dependency calculations are performed by static analysis of the component code, recursively applied to every child component, module, and wire adapter used by the top-level component.

In general, anything that requires code to execute to determine its execution path can't be resolved during static dependency analysis. The following guidelines are an incomplete list of ways to avoid anti-patterns that can prevent a complete dependency analysis of a component.

- Don't use a private property as an input value to a wire adapter.
- Don't use a getter as an input value to a wire adapter when the getter result depends on an instance member or *any* computation.
- Don't use any computed value as an input value to a wire adapter where the value can't be determined without creating and executing the component, or which might be null when the component is instantiated.
- Don't create a wire adapter chain where an earlier wire adapter outputs its results into a function.
- Don't reference an array member from an array that is chained between wire adapters.
- Don't reference an inherited property in an input value to a wire adapter in a subclass.
- Don't create getter functions or properties that match an imported name. If a property name and import name are the same, the static analyzer can't differentiate them, and the imported item can't be primed. For example:

```
import { recordContextQuery } from 'c/myModule';
export default class GetterTest extends LightningElement {
    @api objectApiName;\

    // Don't make the getter name the same as the import name
    // This prevents priming the imported recordContextQuery
    get recordContextQuery() {
        return recordContextQuery(this.objectApiName);
    }
}
```

Tip: See Validate Lightning Web Components for Offline Use to install validation tools that provide support within VS Code for the previous guidelines.

Example and Workaround

Let's consider an example. It's common for a component to use `CurrentPageReference` in its code.

```
import { CurrentPageReference } from 'lightning/navigation';
// ...@wire(CurrentPageReference) pageRef;
```

During initial priming when the app loads, the "current page" can't be known. This means that resources associated with that page — layout metadata, object and field metadata, record data — can't be fully determined in advance. And so the component can't be primed completely.

The workaround in this case, and in general, is to find another way to make these dependencies explicit, instead of implicitly defined by a reference that can't be resolved until runtime. Referencing a specific `recordId`, `recordTypeId`, or `apiName` provides enough information to determine the dependencies without the specific page context.

Determine Online or Offline Status

The Salesforce Mobile app, Field Service Mobile app, and Lightning Web Components generally, don't have a supported mechanism for detecting whether a device is online or not. **This is by design.**

Online vs. offline connectivity is dynamic, and the signal given to the app from the mobile operating system is notoriously unreliable. This lack of a clear signal is because the state of connectivity changes frequently. Connectivity isn't a simple on/off switch. There's a range, from totally offline, to spotty, to slow-but-solid, and all the way up to faster than wired speed in the best circumstances. There's no good way to know if an action that requires a network connection will succeed or fail, except by attempting it.

In general, our design goal is that LWCs work offline first, and treat being offline as a normal condition, not a failure. When a network request doesn't succeed, the condition is handled as gracefully as possible. We recommend your components adhere to this practice.

CHAPTER 6 Use Salesforce Features While Offline

In this chapter ...

- Use GraphQL While Mobile and Offline
- Use Apex While Mobile and Offline
- Use Images in an LWC Offline-Enabled Component
- Upload Images While Offline
- Use Third-Party JavaScript in an LWC Offline-Enabled Component
- Navigation

Although LWC Offline is intended to "just work" when you use features while offline, there are nuances and additional considerations for using some features while offline. This chapter provides details for how to use these features effectively in LWCs to make them offline-ready.

Use GraphQL While Mobile and Offline

GraphQL, often shortened to GQL, is a flexible, powerful query language for accessing record and other data. You can think of GraphQL as a modern equivalent of SQL, the query language for relational databases.

Developers like GraphQL for modern web applications because, in contrast to many REST and CRUD-oriented APIs, GraphQL allows for expressive queries, with features like filtering and scopes, ordering and aggregation, pagination, and relationship traversal to related records. A single query can retrieve many records, and even records of multiple types. Using fewer queries reduces the number of server requests required to load data, which can improve performance. A GQL query can specify precisely and only the fields required for a given component, reducing the amount of data that needs to be transmitted before a page can render.

Salesforce offers several different implementations of GraphQL for use in your apps. Each implementation has an intended context and purpose, and relevant use cases. Learning to use the appropriate implementation, or when you must use a **specific** implementation, is straightforward. See Understand Salesforce GraphQL Implementations on page 137.

For mobile developers building apps that work while offline, however, there is only one implementation that matters: Offline GraphQL. The rest of this chapter provides details of using Offline GraphQL, including important considerations and limitations.

Getting Started

- The fastest way to get up to speed on GraphQL and learn how to use it with the Salesforce Platform is to read the GraphQL API Developer Guide.
- The fastest way to get up to speed on using GraphQL in Lightning web components is to read the Use the GraphQL Wire Adapter chapter of the same developer guide. You'll also want to refer to the `lightning/uiGraphQLApi` Wire Adapters and Functions reference in the *Lightning Web Components Developer Guide*.
- The fastest way to learn how to build offline-ready LWCs with GraphQL is to keep reading this chapter.

If you're new to GraphQL, you should plan to read and absorb all of these resources, in the order listed.

IN THIS SECTION:

Understand Salesforce GraphQL Implementations

Salesforce offers three different ways for LWC developers to use GraphQL in their components. Which one you select for your components will depend on the specific needs of your application.

Feature Limitations of Offline GraphQL

Offline GraphQL uses the same wire adapter mechanism as the standard (online only) LWC wire adapter for GraphQL. You don't change any code to use Offline GraphQL, and your component can be used while online and offline.

Best Practices for Using GraphQL in LWC Offline

There are a number of best practices to be aware of when using GraphQL in your offline-ready components and apps.

Understand Salesforce GraphQL Implementations

Salesforce offers three different ways for LWC developers to use GraphQL in their components. Which one you select for your components will depend on the specific needs of your application.

GraphQL API Endpoint

GraphQL is available using a traditional API endpoint, similar to the UI API. This implementation of GraphQL is the most complete and feature-rich. It also requires you to handle more details yourself: authentication, request submission and response handling, and so on.

The GraphQL API endpoint is accessible to any network-connected client, using any REST-capable tool, framework, or language. It's also the fastest way to "play" with the API; see Quick Starts: GraphQL API to quickly get started using a REST client.

GraphQL Wire Adapter for Lightning Web Components

The GraphQL Wire Adapter lets LWC developers use GraphQL via a wire adapter. Using a wire adapter simplifies the management of your data by using Lightning Data Service (LDS).

Using GraphQL via the wire adapter simplifies making requests and handling responses, and allows your components to leverage other features of LDS, such as caching data and requests for improved performance. The flexibility and expressiveness of GraphQL can enable you to use GQL instead of Apex for advanced data retrieval operations. Preferring LDS and wire adapters over Apex is a best practice, and part of the LWC Data Guidelines.

Offline GraphQL

Offline GraphQL is an implementation of GraphQL built into LWC Offline, which can run on a mobile device, even when it's not connected to the Internet. Offline GraphQL uses the same GraphQL wire adapter as online-only components—you don't have to change your code at all. Indeed, when a mobile device is online, it seamlessly uses the standard (not offline-enabled) GraphQL wire adapter.

However, when a mobile device is offline, and your LWC runs in an LWC Offline-enabled mobile app, Offline GraphQL is used, again, automatically. Offline GraphQL runs client-side, on the mobile device, without a network connection. It uses data and metadata that is already on the device in the Offline Cache, preloaded either by priming or by normal client activity.

Offline GraphQL has a number of limitations you need to be aware of. The standard implementations of GraphQL have access to your org's complete data and metadata, as well as access to the server-side resources of a vast data center. Offline GraphQL has ... your phone.

That might sound limiting, and it is. Offline GraphQL isn't magic. There's no sufficiently advanced GQL query that can load data that's not already present on the device. But with smart data access strategies you *can* make your custom LWCs perform work that *looks* like magic to folks who need to use them in places without network access.

How to Choose a GraphQL Implementation

- For developers working with a framework other than Lightning Web Components, the choice is easy: use the GraphQL API Endpoint. (It's your only option.)
- For quickly experimenting with GraphQL, either as your first experience or just to work out a new GQL query, use the GraphQL API Endpoint and a REST client of your choice. (We document using GraphQL with two different REST clients in Quick Starts: GraphQL API.)
- For LWC developers building components that will be used only while connected to the Internet:
  - Use the GraphQL wire adapter. It's the recommended, natural method for data access in Lightning Web Components, and leverages many additional features to make using GQL easier and more clear in your code.
  - If and only if the GraphQL wire adapter is missing a GraphQL feature offered by the GraphQL API Endpoint, and you must use that feature today, then use the GraphQL API Endpoint. We recommend you isolate this code such that it will be easy to refactor to use the wire adapter when the feature catches up.
- For LWC developers building components that need to work offline, your only option is Offline GraphQL. (Keep reading.)

Feature Limitations of Offline GraphQL

Offline GraphQL uses the same wire adapter mechanism as the standard (online only) LWC wire adapter for GraphQL. You don't change any code to use Offline GraphQL, and your component can be used while online and offline.

While using different code isn't necessary, you must restrict the GraphQL features that you use in your queries. The Offline GraphQL wire adapter supports a subset of the features supported by the standard LWC wire adapter.

Note: This subset grows in every release. A delay is typical in new features, and some features can't be supported while disconnected from Salesforce service.

Let's get to the largest disappointments first. These major features don't work while offline.

- Aggregate queries
- Mutations (data modification)

These major features are partially supported.

- Pagination
  - Pagination is supported on top-level record queries, but not on nested child queries.

Where these features are required for your component or app to function, you must build them yourself, or use other data access mechanisms besides GraphQL.

Data Access (Record Queries)

For "normal" data access, read-only queries that retrieve record data, most features are supported. These features are supported, but with some limitations.

- Most scalar field operators are supported.
  - Boolean operators `and`, `not`, and `or` are supported when the nested predicates (subclauses) use supported features.
  - Geolocation and other compound fields have limited support. Location-based filtering isn't supported.
  - Sometimes picklist and multi-picklist results don't exactly match results from the online implementation, particularly if null values are included in the predicate.
  - Relative date filtering is based on the device locale, not the org locale setting.
- `DisplayValue` for records and `displayValue` for fields are both supported.
- `first` argument to limit query result size is supported.
  - While `first` isn't a pagination feature, it's often used with pagination features that aren't supported.
- `scope` argument:
  - `MINE` is supported for all entities.
  - `ASSIGNEDTOME` is supported for ServiceAppointment.
  - No other scopes are supported by Offline GraphQL.
- `orderBy`
  - Ordering is supported, but it can have a negative impact on performance. Keep in mind where the sorting takes place, that is, your phone. Keep your sort criteria simple, and don't sort more records than necessary.
  - Ordering results by a picklist field sorts by the picklist label, rather than the order of the picklist values defined in Setup.
- Relationships and related record access:
  - Parent-to-child relationships are supported.
  - Child-to-parent relationships are supported.
  - Polymorphic relationship fields such as Owner are supported.

These features aren't supported at this time.

- Compound fields such as `Account.ShippingAddress` aren't supported in selections, predicates, or `orderBy` clauses.

- Semi-join and anti-join filters such as `inq` and `ninq` aren't supported.
- `in` and `nin` operators aren't supported for Date and Date/Time fields.
- Location-based filters aren't supported.
- Fiscal date literals aren't supported in filters.
- Relative date ranges such as "last 30 days" aren't supported in filters.

Metadata

Necessary object metadata, such as custom objects, fields, and layouts, is automatically loaded and cached during priming and online activity. With GraphQL, you can manually query for metadata when the occasion calls for it. When offline, there are some limitations.

- Metadata fetched through GraphQL is cached separately from metadata fetched through the `getObjectInfo` wire adapter. Sometimes they're not perfectly in sync.
- Related list metadata can work, but it's not supported at this time.

Other Feature Considerations

- Record query pagination:
  - The `after` argument to paginate results is supported on top-level record queries. However, the `after` argument to paginate results for nested child relationship queries isn't supported.
  - Cursor field selections aren't supported.
- GraphQL query performance can be suboptimal on complex queries that filter or order by non-indexed fields.
- Queries that reference offline-created (draft) records in the predicate, directly in the query or indirectly through variables, return locally cached results only. The query doesn't make a network request to the server.

Important: Metaschema directives in GraphQL queries were deprecated in Summer '23. However, if your GraphQL query fails prefetch in the Salesforce mobile app, Salesforce Field Service, or Mobile Offline, you must continue to use metaschema directives in your GraphQL query for referential integrity and offline priming functionality. See *Known Issue:* GraphQL query fails prefetch with an "Unknown Field" warning.

SEE ALSO:

Understand Salesforce GraphQL Implementations

Best Practices for Using GraphQL in LWC Offline

There are a number of best practices to be aware of when using GraphQL in your offline-ready components and apps.

In addition to the general best practices detailed in GraphQL Wire Adapter Best Practices, we want to call your attention to two specific practices that are especially important for the Offline GraphQL implementation.

Provide Query and Variables via Getter Function

To make the results of a GraphQL query primable, you must use a specific structure in your code. This structure is specifically required for the GQL query to be discovered and processed by the static analyzer used by the priming subsystem. Do not inline the query with your wire adapter function definition.

For example, this code results in a query that can be analyzed and primed:

```
// BEST PRACTICE
// This GQL query can be primed
@wire(graphql, {
  query: '$myQuery',
  variables: '$myVariables'
})
wiredData;

get myQuery() {
   return gql`query getSomeAccount($recordId: ID) {
              uiapi { query { Account(...) { } } }`;
}
get myVariables() { return { recordId: '...' } }
```

In contrast, this example behaves the same as the preceding code, as long as you're online, but does not result in the wire adapter results being primed. If the required data and metadata isn't primed by some other mechanism, it won't function while you're offline.

```
// ANTI-PATTERN
// This inline GQL query cannot be primed
@wire(graphql, {
  query: gql`query getSomeAccount($recordId: ID) {
             uiapi { query { Account(...) { } } }`,
  variables: {recordId: '...'}
})
wiredData;
```

Delay Query Execution

This best practice is closely related to using getters to provide the GQL query string and variables, and uses the same code structures. It's similarly essential to maximizing the completeness of priming activities performed by the mobile app prior to going offline. The details are described in Delay Query Execution in the *GraphQL API Developer Guide*.

Query Only the Data You Need

You can write a query that satisfies your data needs and gives you exactly what you asked for, nothing more and nothing less. Writing a brief query that satisfies your data requirements makes the return result predictable, while ensuring that your query remains performant.

See Query Only the Data You Need in the GraphQL API Developer Guide for an example on how to use pagination to avoid loading a large number of records in a single query.

Important: In the code example, `totalCount` isn't supported for mobile use cases.

Use Apex While Mobile and Offline

Use Apex-backed wire adapters and imperative Apex in your Lightning web components to call Apex methods in your org.

When you use Apex in an LWC Offline-enabled mobile app, there are considerations to keep in mind so that you make efficient use of network resources, data caching, and handle offline behavior correctly.

First, when the client device is online, Apex-based features of Lightning web components, including Apex continuations, "just work." You can use all of the features as documented in Call Apex Methods in the *Lightning Web Components Developer Guide*.

When a client device is offline, Apex-based features can **read** data that was cached while online, but changes (writing data) can't be saved back to the server. Nor can a change via Apex methods be enqueued as a draft into the Offline Queue. A Lightning web component that uses Apex must be prepared to handle a network connection error as a normal response, for both reading and writing operations.

Important: Before you make plans to reuse existing Apex custom code in your offline features, read important details about offline caching for Apex in Apex Results Are Separate from Other Primed Data.

IN THIS SECTION:

Use Apex in Lightning Web Components While Online

The essentials of using Apex within Lightning web components are described in "Call Apex Methods" in the *Lightning Web Components Developer Guide*. While Apex features behave as documented when a client device is online, there are additional features available within an offline-enabled mobile app.

Enable Caching of Apex Results

To allow results of Apex calls to be saved for offline use, enable caching on Apex methods used in your offline-enabled mobile apps.

Apex Results Are Separate from Other Primed Data

Apex results are saved in the durable store **separately** from data stored by built-in components, modules, and wires that use Lightning Data Service (LDS) to access data.

Understand Apex Behavior While Offline

Additional features that are built into offline-enabled apps allow the app, including Lightning web components and even Apex, to continue to function. Knowing these features, and their limitations, is critical to writing LWCs that function well, even without a connection to the Salesforce service.

Additional Considerations for Apex in an Offline-Enabled Mobile App

The following differences in behavior compared to Apex run from a browser-based connection apply to Apex when used in Lightning web components in an offline-enabled mobile app.

Additional Documentation for Apex in Lightning Web Components

Learn more about how to use Apex, including continuations, from Lightning web components documentation resources.

Use Apex in Lightning Web Components While Online

The essentials of using Apex within Lightning web components are described in "Call Apex Methods" in the *Lightning Web Components Developer Guide*. While Apex features behave as documented when a client device is online, there are additional features available within an offline-enabled mobile app.

To take advantage of these features, you need to know the basics of using Apex within Lightning web components. In particular, there are two different ways of calling an Apex method from a Lightning web component.

IN THIS SECTION:

Reactive Apex Wires

Reading data via a wire adapter is the "natural" way to access data in Lightning web components. To add a read-only Apex method to a Lightning web component, first import the Apex method from the `@salesforce/apex` module, and then use the `@wire` annotation to connect that method to a property or function in your component.

Imperative Apex

Imperative Apex is the more traditional way to call an Apex method, as a network-based API call. Imperative Apex allows you to control exactly when the method is called. You're in control of the invocation, rather than the framework.

Reactive Apex Wires

Reading data via a wire adapter is the "natural" way to access data in Lightning web components. To add a read-only Apex method to a Lightning web component, first import the Apex method from the `@salesforce/apex` module, and then use the `@wire` annotation to connect that method to a property or function in your component.

Read the details in Wire Apex Methods to Lightning Web Components in the *Lightning Web Components Developer Guide*.

The important things to understand about this method of using Apex are:

- `@wire` adapters are **read-only**.
- When you use `@wire` adapters, **you** don't call your Apex method directly. Instead, *the framework* decides when to call it, when it needs the value of the property or function connected to the `@wire`. This can happen more or less frequently than you expect.
- Your Apex method **must not** make server-side changes when called by the `@wire` service.
- The result of calling your Apex method is cached on the client. See important details in Apex Results Are Separate from Other Primed Data.

These points are standard behavior for Lightning web components. They aren't specific to the additional mobile and offline features of an offline-enabled mobile app.

What *is* different is the cache. The mobile app saves the results of `@wire` calls to Apex methods in a *durable store*, instead of an in-memory cache. The durable store is longer-lived than the standard cache used by Lightning web components. The standard cache is designed for performance, rather than offline use. The durable store, in contrast, is designed *specifically* for offline use. It survives client app restarts, and even device restarts. Data in the durable store is available to provide a result for an Apex method call, even when offline. Again, see considerations in Apex Results Are Separate from Other Primed Data.

Imperative Apex

Imperative Apex is the more traditional way to call an Apex method, as a network-based API call. Imperative Apex allows you to control exactly when the method is called. You're in control of the invocation, rather than the framework.

Imperative Apex is more flexible—and less restrictive—than reactive Apex wires. For example, you can use imperative Apex calls to change data on the server. The full details are described in Call Apex Methods Imperatively in the *Lightning Web Components Developer Guide*.

Warning: Salesforce strongly recommends against using imperative Apex for offline use cases.

However, imperative Apex has a significant limitation in an offline-enabled mobile app. **You can't use imperative Apex while offline.** This is by design, since imperative Apex is allowed to change data on the server. There's no way to reconcile what *might* change on the server side with data cached on the client side. This is because the client has no knowledge of the implementation of a server-side Apex method. See Imperative Apex While Offline.

Enable Caching of Apex Results

To allow results of Apex calls to be saved for offline use, enable caching on Apex methods used in your offline-enabled mobile apps.

Annotate the Apex method with `@AuraEnabled(cacheable=true)`, which caches the method results on the client. When you set `cacheable=true`, a method must only retrieve data, it can't mutate (change) data.

- Apex methods used in reactive wires **must** be annotated with `@AuraEnabled(cacheable=true)`, whether you intend to use the results offline or not.
- Apex methods called imperatively only need to be annotated with `@AuraEnabled(cacheable=true)` if you want the results to be available offline.

Additional details of Apex method caching behavior and managing cached results can be found in the *Lightning Web Components Developer Guide*.

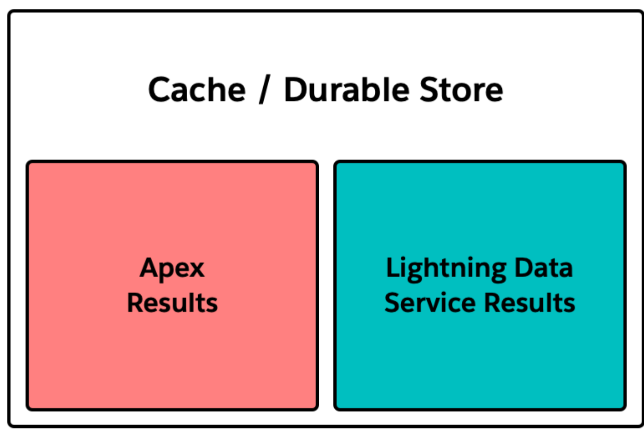
SEE ALSO:

Lightning Web Components Developer Guide: Client-Side Caching of Apex Method Results

Apex Results Are Separate from Other Primed Data

Apex results are saved in the durable store **separately** from data stored by built-in components, modules, and wires that use Lightning Data Service (LDS) to access data.

This is because Apex methods return arbitrary data, as an opaque data blob, rather than returning typed records in expected formats.



Data that's loaded during priming activities, or that's loaded using base components, wire adapters, and other built-in features of Lightning web components, is stored in one consolidated cache. In an offline-enabled mobile app, it's also saved to the durable store for offline use. As you load new data, it's merged into the local store of cached data. This merging includes updating existing items with changes loaded from Salesforce.

The mechanism that makes this possible is Lightning Data Service, or LDS. LDS results include *metadata* about the returned results. This metadata, sometimes referred to as ObjectInfo, allows LDS to treat the results as real Salesforce objects.

Apex requests are different. While Apex methods *can* return nothing more than records, there's nothing that *ensures* this. Apex requests can return arbitrary data, in any format you build in your Apex code. As a result, Apex methods don't return ObjectInfos for the data contained in the response. And without the ObjectInfos for the results, there's no way to treat those results as Salesforce records—and therefore no way to merge them in with records saved locally by LDS.

To simplify the actual implementations, think of LDS results as being records that are indexed by record ID, with each record stored in a separate representation of a Salesforce object. In contrast, Apex results are stored as one big opaque blob per request, indexed on the request URL, including parameters.

While *you* can interpret structured data inside that opaque blob, LDS and the LWC framework can't. If you load the same records with LDS and Apex, those records are stored *separately,* as duplicates. Updates to one copy don't affect the other, unless you fire the Apex request again (to load changes performed via LDS), or tell the framework to reload stale LDS data (to load changes performed via Apex).

This poses a challenge for implementing features that use both the built-in data access features of Lightning Web Components and the fully customizable logic of Apex. Our advice is to use the base Lightning components whenever possible, then use LDS wire adapters, and then if necessary, use Apex. An extensive discussion of this tiered strategy can be found in Data Guidelines in the *Lightning Web Components Developer Guide*.

Note: We recognize that LDS and built in components don't cover every use case, and that there's *always* a place for custom logic performed on the server, especially for transactional operations. We intend to improve the capabilities of LDS over time, including adding advanced capabilities for complex queries and other operations. We advise planning your use of Apex in such a way that you can refactor your data access logic to adopt new LDS features as they become available.

SEE ALSO:

Lightning Web Components Developer Guide: Data Guidelines

Lightning Web Components Developer Guide: Client-Side Caching of Apex Method Results

Understand Apex Behavior While Offline

Additional features that are built into offline-enabled apps allow the app, including Lightning web components and even Apex, to continue to function. Knowing these features, and their limitations, is critical to writing LWCs that function well, even without a connection to the Salesforce service.

The standard Salesforce web interface requires a continuous network connection to the Salesforce service. When a standard Salesforce client, such as a desktop browser, is offline, features that require a connection to the Salesforce service—which is most features—don't work. Since Apex runs on the Salesforce service, this includes *all* Apex-based features of Lightning web components.

An offline-enabled mobile app, in contrast, is designed to continue to function even when no connection to the Salesforce service is available. The offline features of the app aren't magic; data that's not already available on the client device is inaccessible. But carefully designed features, including Lightning web components that use Apex, can continue to run with data that's available locally, on the device, even without a network connection.

IN THIS SECTION:

Apex Wires While Offline

Lightning web components that wire properties or functions to Apex methods continue to provide cached values from the durable store, if available.

Imperative Apex While Offline

Imperative Apex calls always fail when the client device is offline. When using imperative Apex in an offline-enabled mobile app, it's essential to handle the possibility of a network failure error.

Refresh Records Cached in Durable Store While Offline

While it's not possible to retrieve updated data from Salesforce while a client device is offline, it's still possible to request updates when data is known to be stale.

SEE ALSO:

Lightning Web Components Developer Guide: Client-Side Caching of Apex Method Results

Apex Wires While Offline

Lightning web components that wire properties or functions to Apex methods continue to provide cached values from the durable store, if available.

If the result of an Apex method hasn't been retrieved previously and saved to the durable store, an error is returned. The correct way to handle the error depends on whether the Apex method is wired to a property or a function.

When using a property, the wire service either provisions the results to the `<property>.data` property, or returns an error to the `<property>.error` property. Use an `if:true` directive in your component template to check for the presence of each, and render the appropriate output. Changes to `<property>` trigger a re-render of your component with new values.

With a function, the wire service provisions results to the wired function via an object with either an `error` or `data` property. Check for the presence of each in your JavaScript function, and set an appropriate property of your component. This triggers a re-render of your component with new values.

Example code for each of these approaches is available in Wire Apex Methods to Lightning Web Components in the *Lightning Web Components Developer Guide*.

Imperative Apex While Offline

Imperative Apex calls always fail when the client device is offline. When using imperative Apex in an offline-enabled mobile app, it's essential to handle the possibility of a network failure error.

Treat errors as a normal, expected outcome, rather than a failure condition. Provide appropriate feedback to the user, and suggest alternative behavior, rather than treating the situation as unexpected.

```
// apexImperativeMethod.js
import { LightningElement, track } from 'lwc';
import getContactList from '@salesforce/apex/ContactController.getContactList';

export default class ApexImperativeMethod extends LightningElement {
    @track contacts;
    @track error;

    handleLoad() {
        getContactList()
            .then(result => {
                this.contacts = result;
            })
            .catch(error => {
                this.error = error;
            });
    }
}
```

Offline-savvy components *expect* to be offline at times, and know what to do when that happens.

Refresh Records Cached in Durable Store While Offline

While it's not possible to retrieve updated data from Salesforce while a client device is offline, it's still possible to request updates when data is known to be stale.

For example, after updating a record via imperative Apex, you would want any version of that record cached by LDS to be updated. It's straightforward to write code that handles this process when your imperative Apex succeeds while online, but also handles an update error if the client device is offline.

`getRecordNotifyChange()` is used to advertise the need to update cached records that were modified by imperative Apex. It isn't supported at this time. Note that `getRecordNotifyChange(recordIds)` simply *notifies* LDS that the records represented by `recordIds` provided in the function call are known to be stale. It doesn't *share* updated record values with LDS, even if those records are available offline. It's the responsibility of LDS to retrieve the latest values for those records. LDS can only do so when the client is online.

`refreshApex()` is used to request a refresh for data provisioned by an Apex `@wire`. It's usable today and, while online, behaves as expected. However, while `refreshApex(valueProvisionedByApexWireService)` can be called while offline, it requires a network connection to actually succeed, and push a new value to the wired property or function.

Using `getRecordNotifyChange()` and `refreshApex()` to request updates for cached data that are possibly stale is described in Client-Side Caching of Apex Method Results in the *Lightning Web Components Developer Guide*.

SEE ALSO:

Lightning Web Components Developer Guide: Client-Side Caching of Apex Method Results

Lightning Web Components Developer Guide: `@salesforce/apex` in `@salesforce` Modules

Additional Considerations for Apex in an Offline-Enabled Mobile App

The following differences in behavior compared to Apex run from a browser-based connection apply to Apex when used in Lightning web components in an offline-enabled mobile app.

If a quick action uses a wired Apex method, and that quick action is primed at app startup, then the Apex results data can be primed as well, and available offline. The LWC must be statically analyzable for priming to take place. Specifically, input parameters for the wire adapter must be analyzable. For example, if the input parameters are derived from page reference attributes, or from the output of another LDS wire that is also analyzable. See Validate Lightning Web Components for Offline Use for additional details about static analysis. Note that the Apex method will be invoked during priming, possibly many times, to prime results for all possible adapter input parameters.

Apex continuations are supported. However, because continuations tend to be longer running requests, we recommend providing feedback to the user while a continuation is active. Otherwise, they might go offline before a continuation completes, which results in an error.

Additional Documentation for Apex in Lightning Web Components

Learn more about how to use Apex, including continuations, from Lightning web components documentation resources.

In addition to the Call Apex Methods chapter, see the `@salesforce/apex` and `@salesforce/apexContinuations` sections of the `@salesforce` Modules reference in the *Lightning Web Components Developer Guide*.

Use Images in an LWC Offline-Enabled Component

Lightning Web Components supports multiple ways of referencing graphics assets in a component. Not all of these methods work when the component runs offline. LWC Offline supports several methods of referencing images in your offline-ready Lightning web components.

The standard methods for referencing images (and other binary assets) in an LWC are:

- Images uploaded as Files, using the `VersionDataUrl` field of the latest `ContentVersion` related to a particular `ContentDocument` record.
- Content Assets, using the `@salesforce/contentAssetUrl` module.
- Static Resources, using the `@salesforce/resourceUrl` module.

For LWCs with images intended to be used offline, we recommend the first two options. Static resources support offline images, but with limitations.

Note: Support for offline images requires that both your Salesforce org and your mobile app are updated to the Summer '23 release (API 58.0) or later.

IN THIS SECTION:

Use Images Uploaded as Files (ContentDocument) in an LWC

Files are a general mechanism to upload and make binary files, such as images, available in your Salesforce org. Files can be associated with a specific record, which makes them ideal for product photos, images captured during a service call or other transaction, and otherwise adding images to business activities that you track in Salesforce.

Use Images Uploaded as Asset Files

Asset files are the modern alternative to static resources. Asset files are ideal for images that are used throughout your components and apps—for example, user interface elements like icons—or otherwise aren't related to a specific record.

Use Images Uploaded as Static Resources

Static resources are a method for packaging one or more images, stylesheets, or JavaScript files for use within Lightning web components, and other Salesforce customization features.

Image Priming and Offline Considerations

LWC Offline isn't magic. If an image hasn't been primed before you go offline, it can't be displayed while offline. LWC Offline primes image assets that are referenced in component template files, in the `src` attribute of a standard HTML `img` tag.

Use Images Uploaded as Files (`ContentDocument`) in an LWC

Files are a general mechanism to upload and make binary files, such as images, available in your Salesforce org. Files can be associated with a specific record, which makes them ideal for product photos, images captured during a service call or other transaction, and otherwise adding images to business activities that you track in Salesforce.

Files have a complex representation in Salesforce, using multiple standard objects to store the file itself and information about it, including ownership, access controls, and multiple versions of that file. ContentDocument is the primary object and, for the purposes of displaying images in your LWCs, you can reference the binary data of a file through a relationship that is the same for any uploaded file.

The critical elements of an offline-ready implementation are:

- Access the URL of the File's current version through the `@salesforce/schema/ContentDocument.LatestPublishedVersion.VersionDataUrl` related field.
- Provide the image URL via a getter function that parses the record data of the current version.
- Use the getter function for the `src` attribute of an `img` tag in your HTML template.

Important: While there are other methods for referencing images in an LWC, the preceding elements are required for **offline** image access to function. The Komaci static analyzer looks for this specific pattern when determining images to prime. Additionally, your getter function must be **statically** analyzable. If its result can only be determined at runtime, the image can't be primed.

Example: The following example is simple, and uses a hard-coded `ContentDocument` record ID, but it illustrates the details. You can also access named renditions (thumbnails) of the image by adding a `thumb` parameter to the URL.

```
// imageFromContentDocument.js
import { LightningElement, wire } from 'lwc';
import { getFieldValue, getRecord } from 'lightning/uiRecordApi';
import IMAGE_URL_FIELD from
    '@salesforce/schema/ContentDocument.LatestPublishedVersion.VersionDataUrl';

export default class ImageFromContentDocument extends LightningElement {

    @wire(getRecord, {recordId: '069RO0000003FMoYAM', fields: [IMAGE_URL_FIELD] })
    contentDocImage;

    get imageUrl() {
```

```
        return getFieldValue(this.contentDocImage.data, IMAGE_URL_FIELD);
    }

    get resizedImageUrl() {
        return getFieldValue(this.contentDocImage.data, IMAGE_URL_FIELD) +
            '?thumb=THUMB240BY180';
    }
}
```

With the image URL provided by the getter functions in the preceding component JavaScript, referencing the images in the HTML template is just like referencing any image in HTML. Use it in the `src` attribute of an HTML `img` tag.

```
<!-- imageFromContentDocument.html -->
<template>
    <lightning-card>Display an image from
        ContentDocument {imageUrl}</lightning-card>

    <template if:true={contentDocImage}>
        <img src={imageUrl}/>
    </template>

    <lightning-card>Display a resized image from ContentDocument
        {resizedImageUrl}</lightning-card>

    <template if:true={contentDocImage}>
        <img src={resizedImageUrl}/>
    </template>

</template>
```

This feature works in LWC Offline beginning in Spring '23, which is API version 57.0. Be sure to set that minimum API version for any component that references images while offline.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- imageFromContentDocument-meta.xml -->
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>57.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__RecordPage</target>
        <target>lightning__AppPage</target>
        <target>lightning__RecordAction</target>
        <target>lightning__GlobalAction</target>
    </targets>
    <targetConfigs>
        <targetConfig targets="lightning__RecordAction,lightning__GlobalAction">
            <actionType>ScreenAction</actionType>
        </targetConfig>
```

```
    </targetConfigs>
</LightningComponentBundle>
```

SEE ALSO:

User Interface API Developer Guide: Upload Files

Salesforce Object Reference Guide: ContentDocument

Salesforce Object Reference Guide: ContentVersion

Use Images Uploaded as Asset Files

Asset files are the modern alternative to static resources. Asset files are ideal for images that are used throughout your components and apps—for example, user interface elements like icons—or otherwise aren't related to a specific record.

Referencing an image stored in an Asset file is straightforward, and fully documented. We present only a simple example here. See Access Content Asset Files the *Lightning Web Component Developer Guide* for complete details.

The critical elements of an offline-ready implementation are:

- Access the URL of the asset using the `@salesforce/contentAssetUrl` module.
- Provide the image URL via a getter function.
- Use the getter function for the `src` attribute of an `img` tag in your HTML template.

Important: While there are other methods for referencing images in an LWC, the preceding elements are required for **offline** image access to function. The Komaci static analyzer looks for this specific pattern when determining images to prime. Additionally, your getter function must be **statically** analyzable. If its result can only be determined at runtime, the image can't be primed.

Example:

```
// imageFromAssetFile.js
import { LightningElement } from 'lwc';
import ASSET_IMG from '@salesforce/contentAssetUrl/avatars-light-mode';
import ARCHIVE_IMG from '@salesforce/contentAssetUrl/branding-images';

export default class ImageFromAssetFile extends LightningElement {
    get assetUrl() {
        return ASSET_IMG;
    }

    get assetArchiveUrl() {
        return ARCHIVE_IMG + '&pathinarchive=images/logo-large.png';
    }
}
```

Warning: The `assetArchiveUrl` function in the preceding example appends a `pathinarchive` query parameter and value, using a “&” separator. The “&” isn't used in examples in the standard LWC documentation. The need for the “&” separator is inconsistent between desktop and mobile today, and we consider this discrepancy to be a software defect. For now, adding the “&” separator generally works on both desktop and mobile, even though it results in a double “&&” on desktop.

With the image URL provided by the getter functions in the preceding component JavaScript, referencing the images in the HTML template is just like referencing any image in HTML. Use it in the `src` attribute of an HTML `img` tag.

```
<!-- imageFromAssetFile.html -->
<template>
    <lightning-card>
        Display an image directly from an Asset file {assetUrl}
    </lightning-card>

    <template if:true={assetUrl}>
        <img src={assetUrl}/>
    </template>

    <lightning-card>
        Display an image from an archive Asset file {archUrl}
    </lightning-card>

    <template if:true={assetArchiveUrl}>
        <img src={assetArchiveUrl}/>
    </template>

</template>
```

SEE ALSO:

Salesforce Help: Asset Files

Lightning Web Components Developer Guide: Access Content Asset Files

Lightning Web Components Developer Guide: `@salesforce/contentAssetUrl`in `@salesforce` Modules

Use Images Uploaded as Static Resources

Static resources are a method for packaging one or more images, stylesheets, or JavaScript files for use within Lightning web components, and other Salesforce customization features.

Referencing an image stored in a Static Resource is straightforward, and fully documented. We present only a simple example here. See Access Static Resources the *Lightning Web Component Developer Guide* for complete details.

The critical elements of an offline-ready implementation are:

- Access the URL of the resource by importing it using the `@salesforce/resourceUrl` module.
- Provide the image URL via a getter function.
- Use the getter function for the `src` attribute of an `img` tag in your HTML template.

Important: While there are other methods for referencing images in an LWC, the preceding elements are required for **offline** image access to function. The Komaci static analyzer looks for this specific pattern when determining images to prime. Additionally, your getter function must be **statically** analyzable. If its result can only be determined at runtime, the image can't be primed.

Example:

```
// imageFromStaticResource.js
import { LightningElement } from 'lwc';
import TRAILHEAD_LOGO from '@salesforce/resourceUrl/trailhead_logo';
```

```
export default class ImageFromStaticResource extends LightningElement {
    get trailheadLogoUrl() {
        return TRAILHEAD_LOGO;
    }
}
```

Warning: LWC Offline doesn't support archive static resources at this time. While you can upload each image as a separate static resource, we recommend that you use Content Assets, which do support archive files, for collections of related images. See Use Images Uploaded as Asset Files.

With the image URL provided by the getter function in the preceding component JavaScript, referencing the image in the HTML template is just like referencing any image in HTML. Use it in the `src` attribute of an HTML `img` tag.

```
<!-- imageFromStaticResource.html -->
<template>
    <lightning-card>
        Display an image directly from a static resource: {trailheadLogoUrl}
    </lightning-card>

    <template if:true={trailheadLogoUrl}>
        <img src={trailheadLogoUrl}/>
    </template>

</template>
```

SEE ALSO:

Salesforce Help: Static Resources

Lightning Web Components Developer Guide: Access Static Resources

Lightning Web Components Developer Guide: `@salesforce/resourceUrl`in `@salesforce` Modules

Image Priming and Offline Considerations

LWC Offline isn't magic. If an image hasn't been primed before you go offline, it can't be displayed while offline. LWC Offline primes image assets that are referenced in component template files, in the `src` attribute of a standard HTML `img` tag.

Note: Only the `img` tag is supported at this time. Images referenced in other ways or tags aren't primed and won't display sometimes, even when online.

Primed images are stored locally on the mobile device, in a binary durable store (cache). Images stored in individual files are primed and cached individually. Images stored in archive asset files are accessible, but be aware that the entire archive is primed and cached. Updates to images in archive files require reloading the entire archive. Archive static resources aren't supported at this time.

In Summer '23, primed images aren't automatically purged from the offline cache when they get stale. Be mindful of the size of your images or archive asset files, and the space they take up locally on devices. LWC Offline-enabled apps can provide other methods of purging stale images. Check the documentation for your target mobile app.

The binary contents of an item referenced in an `img` tag isn't validated. It's up to you to ensure that referenced files are valid images. Support for specific image formats is dependent on the capabilities of the web view, which is provided by the operating system. Providing an unsupported or non-image file to an `img` tag is an HTML error and can cause unpredictable behavior. As the joke goes, don't do that. Safe formats are the usual web image formats: GIF, JPEG, PNG, and so on. When in doubt, test specific image formats on your specific supported mobile devices and operating system versions.

Upload Images While Offline

Upload files, such as images, to Salesforce, even when your mobile client is offline. For example, upload photos of equipment installed during a service call, even if there's no Internet service available. Images upload when your mobile device regains network service.

Important: This feature is available only in Salesforce mobile apps, such as Salesforce Mobile App Plus and Field Service Mobile. This feature depends on functionality built into these mobile apps. While LWCs that use this feature don't emit errors when used on desktop, they don't function as intended, either.

Your organization must purchase and license Salesforce Mobile App Plus in order to use this feature in the Salesforce mobile app. Contact your Salesforce sales rep for more information.

IN THIS SECTION:

Understand File Uploads in Salesforce

It's simple to upload files and attach them to other records in the Salesforce user interface, but a great deal takes place behind the scenes. File uploads have a complex representation in Salesforce. This complexity makes it challenging to work with file uploads programmatically in LWC code, especially in a way that works while a mobile device is offline.

Image Upload Basics

Uploading an image from an LWC using features supported offline is a two-step process. First, use `createContentDocumentAndVersion` to upload the image file. This adapter creates `ContentDocument` and `ContentVersion` records for the file upload. After the `ContentDocument` and `ContentVersion` exist, use `createRecord` to create the `ContentDocumentLink` record that relates the image upload to the record you want to attach it to.

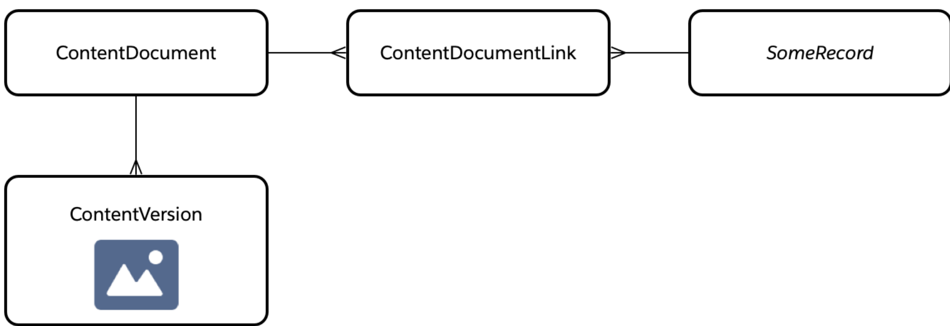
Image Upload Example

Here's a quick example of uploading an image to Salesforce using the `createContentDocumentAndVersion` adapter. This technique works on mobile devices, whether they're online or offline. This technique works only in Salesforce mobile apps.

Understand File Uploads in Salesforce

It's simple to upload files and attach them to other records in the Salesforce user interface, but a great deal takes place behind the scenes. File uploads have a complex representation in Salesforce. This complexity makes it challenging to work with file uploads programmatically in LWC code, especially in a way that works while a mobile device is offline.

For the purposes of thinking about uploading files from an LWC, we can simplify the representation to four related objects.



Warning: This diagram oversimplifies file uploads in Salesforce. It omits ownership, sharing, and other features supported by additional objects and APIs. For more details, including available fields and usage notes, start with ContentDocument, ContentVersion, and ContentDocumentLink in the *Salesforce Object Reference*.

An uploaded, attached, or generated file is represented primarily by the `ContentDocument` object. However, this object holds no file data. Instead, versions of an uploaded file are stored as `ContentVersion` objects. With this structure, you can update a file by uploading a new version. Each upload creates another `ContentVersion` record, and this record is where the binary file data for the upload is stored.

That record represents the file. To associate an uploaded file to another record—for example, to attach photos of a vehicle to a car rental agreement record—the `ContentDocumentLink` object is used to link the two records together. A `ContentDocumentLink` object represents the relationship between a `ContentDocument` and any other record. Associate an uploaded file with any number of other records by creating the appropriate `ContentDocumentLink` records that join them.

When online, the LWC code for creating a file upload, while non-trivial, is straightforward, using standard Lightning Data Service (LDS) adapters like `createRecord`.

Supporting offline uploads from mobile clients is more challenging. Not only is it hard to manage the various relationships between records, but there's also the challenge of holding binary file data in an offline store for later upload. To manage these technical issues, use the dedicated, mobile-specific file upload adapter: `createContentDocumentAndVersion`.

Image Upload Basics

Uploading an image from an LWC using features supported offline is a two-step process. First, use `createContentDocumentAndVersion` to upload the image file. This adapter creates `ContentDocument` and `ContentVersion` records for the file upload. After the `ContentDocument` and `ContentVersion` exist, use `createRecord` to create the `ContentDocumentLink` record that relates the image upload to the record you want to attach it to.

Image Upload Example

Here's a quick example of uploading an image to Salesforce using the `createContentDocumentAndVersion` adapter. This technique works on mobile devices, whether they're online or offline. This technique works only in Salesforce mobile apps.

This component is intended to be used as a quick action in a record context, for example, added to a record detail page. Uploaded images are attached to the associated record, and they can be viewed in the **Notes & Attachments** related items panel for that record.

```
<!-- fileUpload.html -->
<template>
    <h1>File Upload</h1>

    <!-- File selection controls. Always displayed.
         Set `accept="*/*"` to allow uploads of any type of file. -->
    <div>
        <lightning-input
                type="file"
                name="fileUploader"
                label="Select file to upload"
                multiple="false"
                accept="image/*"
                onchange={handleFilesInputChange}
            >
            </lightning-input>
```

```
        </div>

    <!-- If a file is selected, display additional input controls. -->
    <div if:true={fileName}>

        <!-- Show the filename (read-only) -->
        <p>Selected file:</p>
        <p>{fileName}</p>

        <!-- Form fields for upload details -->
        <div class="inputs">
            <lightning-input
                type="text"
                label="Title"
                value={titleValue}
                onchange={handleTitleInputChange}
            ></lightning-input>
            <lightning-input
                type="text"
                label="Description"
                value={descriptionValue}
                onchange={handleDescriptionInputChange}
            ></lightning-input>
        </div>

        <!-- Button to actually do the upload (enqueued as a draft) -->
        <button
            class="slds-button slds-button_brand slds-var-m-top_medium"
            disabled={uploadingFile}
            onclick={handleUploadClick}
        >
            <label>Upload</label>
        </button>
    </div>

    <!-- If there are errors, show them here -->
    <div if:true={errorMessage}>
        <lightning-card title="Error">
            <div class="card-body">{errorMessage}</div>
        </lightning-card>
    </div>
</template>
```

The user interface has three main sections.

- A local file selection widget, which is always displayed.
- A selected file info panel, which is displayed only when there's a file selected. This panel also contains an **Upload** button that triggers the file upload to Salesforce.
- An error messages panel, which is displayed only when there's an error with an upload.

This template is standard markup for a simple widget. All the magic happens on the other side of those four `onchange` attributes, and the handler functions that perform actions when the controls are used.

Here's the component's JavaScript implementation.

```
// fileUpload.js
import { LightningElement, api, track, wire } from "lwc";
import { ShowToastEvent } from "lightning/platformShowToastEvent";
import {
  createContentDocumentAndVersion,
  createRecord,
} from "lightning/uiRecordApi";
// Imports for forced-prime ObjectInfo metadata work-around
import { getObjectInfos } from "lightning/uiObjectInfoApi";
import CONTENT_DOCUMENT from "@salesforce/schema/ContentDocument";
import CONTENT_VERSION from "@salesforce/schema/ContentVersion";
import CONTENT_DOCUMENT_LINK from "@salesforce/schema/ContentDocumentLink";

export default class FileUpload extends LightningElement {
  @api
  recordId;

  @track
  files = undefined;

  @track
  uploadingFile = false;

  @track
  titleValue = "";

  @track
  descriptionValue = "";

  @track
  errorMessage = "";

  // Object metadata, or "ObjectInfo", is required for creating records
  // while offline. Use the getObjectInfos adapter to "force-prime" the
  // necessary object metadata. This is a work-around for the static analyzer
  // not knowing enough about the file object schema.
  @wire(getObjectInfos, {
    objectApiNames: [ CONTENT_DOCUMENT, CONTENT_VERSION, CONTENT_DOCUMENT_LINK ],
  })
  objectMetadata;

  // Getter used for local-only processing. Not needed for offline caching.
  // eslint-disable-next-line
@salesforce/lwc-graph-analyzer/no-getter-contains-more-than-return-statement
  get fileName() {
    // eslint-disable-next-line
@salesforce/lwc-graph-analyzer/no-unsupported-member-variable-in-member-expression
    const file = this.files && this.files[0];
    if (file) {
      return file.name;
    }
    return undefined;
  }
```

```
// Input handlers
handleFilesInputChange(event) {
  this.files = event.detail.files;
  this.titleValue = this.fileName;
}

handleTitleInputChange(event) {
  this.titleValue = event.detail.value;
}

handleDescriptionInputChange(event) {
  this.descriptionValue = event.detail.value;
}

// Restore UI to default state
resetInputs() {
  this.files = [];
  this.titleValue = "";
  this.descriptionValue = "";
  this.errorMessage = "";
}

// Handle uploading a file, initiated by user clicking Upload button
async handleUploadClick() {
  // Make sure we're not already uploading something
  if (this.uploadingFile) {
    return;
  }

  // Make sure we have something to upload
  const file = this.files && this.files[0];
  if (!file) {
    return;
  }

  try {
    this.uploadingFile = true;

    // Create a ContentDocument and related ContentDocumentVersion for
    // the file, effectively uploading it
    const contentDocumentAndVersion =
      await createContentDocumentAndVersion({
        title: this.titleValue,
        description: this.descriptionValue,
        fileData: file,
      });
      console.log("ContentDocument and ContentDocumentVersion records created.");

    // If component is run in a record context (recordId is set), relate
    // the uploaded file to that record
    if (this.recordId) {
      const contentDocumentId = contentDocumentAndVersion.contentDocument.id;
```

```
        // Create a ContentDocumentLink (CDL) to associate the uploaded file
        // to the Files related list of the target recordId
        await this.createContentDocumentLink(this.recordId, contentDocumentId);
      }

      // Status and state updates
      console.log("File upload created and enqueued.");
      this.notifySuccess();
      this.resetInputs();
    } catch (error) {
      console.error(error);
      this.errorMessage = error;
    } finally {
      this.uploadingFile = false;
    }
  }

  // Create link between new file upload and target record
  async createContentDocumentLink(recordId, contentDocumentId) {
    await createRecord({
      apiName: "ContentDocumentLink",
      fields: {
        LinkedEntityId: recordId,
        ContentDocumentId: contentDocumentId,
        ShareType: "V",
      },
    });
    console.log("ContentDocumentLink record created.");
  }

  notifySuccess() {
    this.dispatchEvent(
      new ShowToastEvent({
        title: "Upload Successful",
        message: "File enqueued for upload.",
        variant: "success",
      })
    );
  }
}
```

For the purpose of explanation, we can divide the implementation into these four sections.

- Import statements
- State tracking
- Simple convenience functions and change handlers
- The file upload handler

Import Statements

The only thing interesting about the import statements is the API function, `createContentDocumentAndVersion`. The file upload discussion describes how to use this function.

State Tracking

This component's state tracking consists of one `@api` public attribute and five `@track` internal state attributes.

- `recordId` is public and allows the component to receive a record context. This context is used to associate (attach) files that are uploaded to the record from which the component is launched. For example, to attach photos of equipment installed to the Service Appointment record of a technician's visit.
- `files` holds the currently selected local file prior to being uploaded. This variable is used to hold a file locally while the Title and Description are edited. It's an array so that, with some minor code changes, you can upload multiple files at a time.
- `titleValue` and `descriptionValue` hold the form field values for editing via the form fields.
- `uploadingFile` indicates active processing and is used to manage the **Upload** button's enabled or disabled state.
- `errorMessage` holds messages about any errors that occur when the actual upload is attempted.

Convenience and Handler Functions

- The `fileName` getter is used locally only. It's not relevant to or used for analysis of what to prime, so it's exempt from the "simple getters only" rule.
- `resetInputs` resets the form fields and state after a successful upload.
- `handleFilesInputChange`, `handleTitleInputChange`, and `handleDescriptionInputChange` each update internal state values, in response to user changes on the form.

The code for each of these handlers is short, simple, and reasonably self-explanatory. They're common for any LWC that handles user input via a form. We'll talk about how they're used in the next section, but we'll leave these few lines of implementation code for you to read through yourself.

File Upload Handler Functions

The `handleUploadClick` and `createContentDocumentLink` functions together perform all of the work to upload a file to Salesforce, and link that file to an associated record. Both functions are defined as asynchronous using the `async` keyword.

`handleUploadClick` handles the user interface event (clicking the **Upload** button), and also creates the file upload. `createContentDocumentLink` is a utility function that creates the relationship between the file upload and the "owning" record. These functions are fairly different in how they work, so they're described separately.

`handleUploadClick` is called when the user clicks the **Upload** button, which can only happen after they select a local file to be uploaded. It nevertheless begins by checking for a couple of situations where the upload can't succeed:

- If an upload is already in progress, don't start a new one.
- If there's no actual file to upload, don't try to upload a nonexistent file.

The user interface state *should* prevent these situations by disabling the **Upload** button when either of those conditions are true. However, a well-written function verifies its inputs. These checks ensure that you don't cause an error if these important assumptions aren't correct.

The actual upload processing takes place within a `try` block because all data mutations have the opportunity to fail, especially when you're allowing for them to occur while offline. The first part of uploading a file is creating a file upload with a call to the new API function.

```
const contentDocumentAndVersion =
  await createContentDocumentAndVersion({
    title: this.titleValue,
    description: this.descriptionValue,
    fileData: file,
  });
```

This one call creates two related records. One record is a `ContentDocument` representing the file, including the name and description. The second is a related `ContentVersion` record that holds the file data and represents the current version of the uploaded file.

This one call does a lot of work, including creating the relationship between the two records. While you can achieve the same end result using the `createRecord` adapter, you can do that only while online. Creating and preserving the relationship between the two isn't possible using `createRecord` while offline, mostly due to the complexity of the representation of files in Salesforce. `createContentDocumentAndVersion` abstracts that complexity, making file uploads as simple as the preceding snippet, which is just one line of code, wrapped for readability.

`createContentDocumentAndVersion` creates a file upload, but it does *not* associate that uploaded file with the "owning" record for the record context (if any). After it completes the upload (notice the `await` keyword before the call), we verify that we have a record context (`recordId`). If so, call the `createContentDocumentLink` helper function to create that association, in the form of a `ContentDocumentLink` record.

If `handleUploadClick` is exotic for using a mobile only API function, `createContentDocumentLink` is boring, using `createRecord`, a staple of LWC code since the framework's release.

```
async createContentDocumentLink(recordId, contentDocumentId) {
  await createRecord({
    apiName: "ContentDocumentLink",
    fields: {
      LinkedEntityId: recordId,
      ContentDocumentId: contentDocumentId,
      ShareType: "V",
    },
  });
```

This code is another one-liner when you unwrap it. The trick is knowing enough about the representation of files and their relationship to other object types in Salesforce. In this case, it's knowing that `ContentDocumentLink` represents a relationship between an uploaded file and another record, and knowing which fields to stick the relevant record IDs into.

Tip: `ShareType: "V"` might seem a bit mysterious, but it's simple and not particularly relevant. It sets the sharing level to view-only. See ContentDocumentLink in the *Object Reference for the Salesforce Platform* for details.

Where's the "Offline" Part?

You just finished looking at it. And it looks a lot like regular LWC code for an online-only mobile feature. The only thing new is the `createContentDocumentAndVersion` adapter. There's nothing offline-specific about the code here—it works fine while you're online, too. The offline details are behind the scenes. Follow the LWC Offline guidelines for optimizing priming, and you're good to go.

Use Third-Party JavaScript in an LWC Offline-Enabled Component

Use static resources to provide access to third-party JavaScript libraries in your Lightning web components. To enable JavaScript libraries in static resources to be used while offline, follow these guidelines.

Loading JavaScript libraries stored in a static resource is straightforward, and fully documented. We present only a simple example here. See Use Third-Party JavaScript Libraries the *Lightning Web Component Developer Guide* for complete details.

Important: LWC Offline doesn't support archive static resources at this time. This limitation poses a challenge for libraries that consist of many separate JavaScript files. While you can create a separate static resource for each file and load them individually, that's tedious. Look for a merged, concatenated, or minimized version of your JavaScript library; transformed versions of libraries are often provided by the developer for performance and ease of deployment.

The critical elements of an offline-ready implementation are:

- Access the URL of the resource by importing it using the `@salesforce/resourceUrl` module.
- Load the library using the `loadScript()` function of the `platformResourceLoader` module, and then;
- Call your library's entry point, initialization or factory function, or otherwise start using the library in a `then()` block chained from your `loadScript()` call.

Important: While there are other methods for loading JavaScript libraries in an LWC, the preceding elements are required for **offline** access to function. Static resources aren't primed in advance; users must load (view) a component using the static resource before going offline. This behavior will change in a future release.

Example:

```
// javascriptFromStaticResource.js
import { LightningElement } from 'lwc';
import { loadScript } from 'lightning/platformResourceLoader';
import myLib from '@salesforce/resourceUrl/myLib';

export default class JavascriptFromStaticResource extends LightningElement {
    loadScript(this, myLib)
    .then(() => {
        let result = myLib.myFunction(2,2);
    });
}
```

If you must load multiple separate JavaScript files, wrap them in a Promise, and call the initialization function after all of your calls to `loadScript()` have resolved. For example:

```
Promise.all([
    loadScript(this, resourceName1),
    loadScript(this, resourceName2),
    loadScript(this, resourceName3)
]).then(() => {
    // Start using the library here
});
```

SEE ALSO:

Salesforce Help: Static Resources

Lightning Web Components Developer Guide: Use Third-Party JavaScript Libraries

Lightning Web Components Developer Guide: `@salesforce/resourceUrl`in `@salesforce` Modules

Lightning Web Components Developer Guide: Platform Resource Loader

Navigation

Build navigation for Lightning web components.

When we talk about "navigation," we can broadly separate things into two categories: things that *appear on the screen*—buttons, tabs, links, and so on—and *what actually happens* when someone interacts with those things. Call these two categories *navigation user experience* and *navigation actions*. These terms are loose, not specific to Salesforce, mobile apps, or Lightning web components, and we use them informally here.

IN THIS SECTION:

Navigation User Experience

There are a variety of base components available to design the visual user interface that users tap or click to move around in your app.

Base Components with Built-In Navigation Actions

Use base components that have automatic or built-in navigation actions.

Programmatic Navigation Actions

Some features require more complicated navigation designs. For the most complete control of your user interface and navigation scheme, define navigation actions using JavaScript.

Navigation User Experience

There are a variety of base components available to design the visual user interface that users tap or click to move around in your app.

See Create Mobile-Ready Components in the *Lightning Web Components Developer Guide* for mobile-specific basics, and the Component Reference for a complete catalog of available base components, including usage documentation.

SEE ALSO:

Create Mobile Components

Lightning Web Components Developer Guide: Component Reference

Base Components with Built-In Navigation Actions

Use base components that have automatic or built-in navigation actions.

The following base components support adding navigation controls to your own components. These components provide ways to open another page, or otherwise perform URL-based actions. You supply the URL, or text containing linkable items, and the component takes care of the rest.

Base Component	Details
`lightning-breadcrumb`	Behaves like a hyperlink if a URL is provided via the `href` attribute.
`lightning-carousel-image`	Behaves like a hyperlink if a URL is provided via the `href` attribute.
`lightning-click-to-dial`	Displays a formatted phone number as click-to-dial enabled or disabled for Open CTI and Voice.
`lightning-formatted-address`	Displays a formatted address with a link to the given location on Google Maps.
`lightning-formatted-email`	Displays an email as a hyperlink with the `mailto:` URL scheme.
`lightning-formatted-phone`	Displays a phone number as a hyperlink with the `tel:` URL scheme.
`lightning-formatted-rich-text`	Creates hyperlinks in rich text automatically for linkable text and email addresses.
`lightning-formatted-text`	URLs and email addresses are displayed as hyperlinks when you specify the `linkify` attribute.
`lightning-formatted-url`	Displays a URL as a hyperlink.
`lightning-pill`	Behaves like a hyperlink if a URL is provided via the `href` attribute.

Base Component	Details
`lightning-vertical-navigation`	Behaves like a hyperlink if a URL is provided via the `href` attribute.

Note: Some of these components have limitations when used in an offline-enabled mobile app. See Base Components Support for details.

There are additional base components that can have navigation actions attached to them, such as buttons and tabs. These components require you to add click handlers or other functionality in the form of JavaScript code. See Programmatic Navigation Actions.

SEE ALSO:

Lightning Web Components Developer Guide: Component Reference

Programmatic Navigation Actions

Some features require more complicated navigation designs. For the most complete control of your user interface and navigation scheme, define navigation actions using JavaScript.

In Lightning web components, navigation actions are built using the *navigation service*, provided by the `lightning/navigation` module. See Navigate to Different Page Types in the *Lightning Web Components Developer Guide* to get started.

This module is supported for use in LWC Offline-enabled mobile apps. Each different Salesforce mobile app has different features, and each implements support for the navigation service independently. As a consequence, there are some differences in available and supported navigation actions. See the documentation for your specific mobile apps for more details.

IN THIS SECTION:

Navigation Actions in the Salesforce Mobile App

Use these supported programmatic navigation actions in your Lightning web components intended for use in Salesforce Mobile App Plus.

Navigation Actions in the Field Service Mobile App

Use these supported programmatic navigation actions in your Lightning web components intended for use in the Field Service Mobile app.

Common Navigation Actions

These common navigation actions aren't specific to Salesforce. Depending on your situation, use the LWC navigation service, or in some special cases use standard JavaScript code techniques.

SEE ALSO:

Lightning Web Components Developer Guide: Basic Navigation

Lightning Web Components Developer Guide: Navigate to Different Page Types

Lightning Web Components Developer Guide: PageReference Types

Navigation Actions in the Salesforce Mobile App

Use these supported programmatic navigation actions in your Lightning web components intended for use in Salesforce Mobile App Plus.

Each Salesforce mobile app implements support for the navigation service independently, which results in some differences in available navigation actions. The following PageReference types are supported by the LWC navigation service when used in the Salesforce Mobile App Plus mobile app.

- `standard__quickAction`
- `standard__webPage`

You can implement a surprising number of different navigation actions with these PageReference types. The following are examples of navigation actions, and the PageReference used to implement them.

Navigate to a Quick Action

Create a navigation action that opens an LWC-based quick action.

```
{
    "type": "standard__quickAction",
    "attributes": {
        "actionName": `objectApiName.actionApiName` },
        "state": {
            "recordId": "<recordId>",
            "objectApiName": "<objectApiName>"
    }
}
```

`objectApiName.actionApiName` represents the name of the quick action (*`actionApiName`*), and the sObject that it's defined on (*`objectApiName`*). The `state` object provides a way to pass data into the target component. In this example, `recordID` and `objectApiName` are public properties defined in the *`objectApiName.actionApiName`* quick action's JavaScript code.

Open Salesforce Mobile App via Deep Link

Create a navigation action that leaves Mobile App Plus, and opens a specific page in the Salesforce mobile app.

```
{
    "type": "standard__webPage",
    "attributes": {
        "url": "salesforce1://sObject/001D000000Jwj9v/view"
    }
}
```

There's a wide range of targets available for deep linking into the Salesforce Mobile app. See Configure Deep Linking for the Salesforce Mobile App in the Salesforce Help for available URL formats.

Open the Field Service Mobile App via Deep Link

Create a navigation action that leaves Offline App Plus, and opens a specific page in the Field Service mobile app.

```
{
    "type": "standard__webPage",
    "attributes": {
        "url": `com.salesforce.fieldservice://v1/sObject/${this.recordId}/details`
    }
}
```

There's a wide range of targets available for deep linking into the Field Service mobile app. See Deep Linking Schema for the Field Service Mobile App in the *Field Service Developer Guide* for available URL formats.

Open Web Page

Create a navigation action that opens a screen that displays an external web page.

```
{
    "type": "standard__webPage",
    "attributes": {
        "url": "https://salesforce.com"
    }
}
```

Open Email App

Create a navigation action that opens the device's native email client and pre-fills the addressee and subject lines.

```
{
    "type": "standard__webPage",
    "attributes": {
        "url": "mailto:help@AcmeSupport.com?subject=Help with Asset"
    }
}
```

Open Phone App

Create a navigation action that opens the device's native phone app, and dials a phone number.

```
{
    "type": "standard__webPage",
    "attributes": {
        "url": "tel:123-456-7890"
    }
}
```

Open Message App

Create a navigation action that opens the device's native SMS or message app, and pre-fills the recipient phone number.

```
{
    "type": "standard__webPage",
    "attributes": {
        "url": "sms:12345678"
    }
}
```

SEE ALSO:

Salesforce Help: Configure Deep Linking for the Salesforce Mobile App

Navigation Actions in the Field Service Mobile App

Use these supported programmatic navigation actions in your Lightning web components intended for use in the Field Service Mobile app.

Each Salesforce mobile app implements support for the navigation service independently, which results in some differences in available navigation actions. The following PageReference types are supported by the LWC navigation service when used in the Field Service Mobile app.

- `standard__webPage`

You can implement a surprising number of different navigation actions with this PageReference type. The following are examples of navigation actions, and the PageReference used to implement them.

Navigate from LWC to a Native Screen via Deep Link

Create a navigation action that moves from a LWC to a screen native to the Field Service mobile app.

```
{
    "type": "standard__webPage",
    "attributes": {
        "url": `com.salesforce.fieldservice://v1/sObject/${this.recordId}/details`
    }
}
```

There's a wide range of targets available for deep linking into the Field Service mobile app. See Deep Linking Schema for the Field Service Mobile App in the *Field Service Developer Guide* for available URL formats.

Navigate to a Quick Action via Deep Link

Create a navigation action that opens a quick action, including quick actions built with LWCs.

```
{
    "type": "standard__webPage",
    "attributes": {
        "url":
`com.salesforce.fieldservice://v1/sObject/${this.recordId}/quickaction/<api_name>`
    }
}
```

Open Salesforce Mobile App via Deep Link

Create a navigation action that leaves the Field Service mobile app, and opens a specific page in the Salesforce mobile app.

```
{
    "type": "standard__webPage",
    "attributes": {
        "url": "salesforce1://sObject/WorkOrder/home"
    }
}
```

See Configure Deep Linking for the Salesforce Mobile App for available URL formats.

Open Web Page

Create a navigation action that opens a screen that displays an external web page.

```
{
    "type": "standard__webPage",
    "attributes": {
```

```
        "url": "https://salesforce.com"
    }
}
```

Open Email App

Create a navigation action that opens the device's native email client and pre-fills the addressee and subject lines.

```
{
    "type": "standard__webPage",
    "attributes": {
        "url": "mailto:help@AcmeSupport.com?subject=Help with Asset"
    }
}
```

Open Phone App

Create a navigation action that opens the device's native phone app, and dials a phone number.

```
{
    "type": "standard__webPage",
    "attributes": {
        "url": "tel:123-456-7890"
    }
}
```

Open Message App

Create a navigation action that opens the device's native SMS or message app, and pre-fills the recipient phone number.

```
{
    "type": "standard__webPage",
    "attributes": {
        "url": "sms:12345678"
    }
}
```

SEE ALSO:

Field Service Developer Guide: Deep Linking Schema for the Field Service Mobile App

Common Navigation Actions

These common navigation actions aren't specific to Salesforce. Depending on your situation, use the LWC navigation service, or in some special cases use standard JavaScript code techniques.

Important: Differences between desktop and mobile, and between different mobile applications, can affect the behavior of these common navigation actions. Test your navigation thoroughly, on every platform and device onto which you plan to deploy your components.

Open an Arbitrary URL

Navigate to another web page or URL. The equivalent of clicking a link, or entering a URL into the browser location field.

Don't use `window.open()` to open or navigate to a new URL. Instead, use the navigation service to navigate to the URL using a web page PageReference.

```
import { NavigationMixin } from "lightning/navigation";
// ...

  navigateToDsc() {
    this[NavigationMixin.Navigate]({
      type: "standard__webPage",
      attributes: {
        "url": "https://developer.salesforce.com"
      },
    });
  }
```

In addition to standard URLs, you can use the navigation service to open other apps or features. See PageReference Types and the deep link documentation for your mobile apps for additional details on the kinds and format of special URL types.

Close a Modal Quick Action Panel

Close a modal panel opened by a quick action, usually to dismiss or cancel the action.

When used to close or cancel a quick action, this is a special case of the Go Back navigation action. As a result, it's tempting to use the built-in `window.history.back()` JavaScript function. This works in some, but not all, contexts.

The correct approach in Lightning Web Components is to fire the `CloseActionScreenEvent` event, and let the framework take care of closing the panel, disposing of framework resources, and so on. For example:

```
import { CloseActionScreenEvent } from "lightning/actions";
// ...

  handleCancelClick(clickEvent) {
    // Close the modal window
    this.dispatchEvent(new CloseActionScreenEvent());
  }
```

Not all Salesforce mobile apps support the `CloseActionScreenEvent` event. For those mobile apps, use `window.history.back()` as a work-around.

Go Back

Navigate back to the previous page, the equivalent of clicking the browser Back button.

This navigation action is so common, it's built into all browsers. As a result, it generally doesn't have on screen user interface elements in most web apps. However, Salesforce mobile apps don't have a Back button in the standard user interface. If you want to provide a button or other UI element to navigate backwards, you'll need to build it yourself.

The standard method for doing so is to use the `window.history.back()` JavaScript function that's available in most browser containers. This function depends on the history mechanism built into the browser, or the web view of a mobile app, where it requires

explicit support. As a result, `window.history.back()` can behave differently across different browsers and mobile apps. It's not supported on all Salesforce mobile apps.

SEE ALSO:

Lightning Web Components Developer Guide: PageReference Types

Salesforce Help: Configure Deep Linking for the Salesforce Mobile App

Field Service Developer Guide: Deep Linking Schema for the Field Service Mobile App

CHAPTER 7 Development Tools and Processes

In this chapter ...

- Understand the Mobile Development Cycle
- Set Up Your Development Environment
- Preview Lightning Web Components on Mobile
- Validate Lightning Web Components for Offline Use
- Develop Offline-Ready LWCs with the LWC Offline Test Harness
- Debug Your Components with Virtual Device Builds
- Customize the Offline Experience for the Salesforce Mobile App

Set up and use your development tools for the most efficient developer experience while building mobile and offline LWCs.

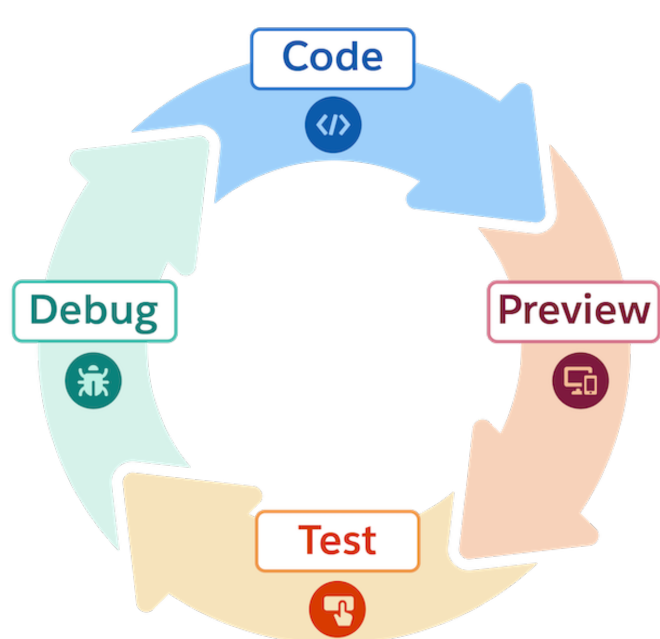
Developing LWCs and apps for use in Salesforce mobile apps uses all the same developer tools and processes that developing LWCs for desktop use. It also poses special challenges, especially in the areas of testing and debugging. This chapter describes a number of additional tools and processes designed specifically for developing mobile LWCs.

Understand the Mobile Development Cycle

Learn the basics of the development cycle for mobile components, including the essential tasks and tools that you need to be productive while building LWCs for use on mobile devices.

Everyone's development process is different, in ways large and small. But no matter how different, there are common aspects of the active development cycle — the "lather, rinse, repeat" — that every productive developer uses. Developing mobile components is no different in that regard, but there are some differences in the details. In particular, there are unique challenges in examining code during development that require some specialized tools and techniques.

The Active Development Cycle



Code

You know what this part is. Actually writing code in your development editor of choice. Coding is the fun part of every developer's job, and the more time you can spend doing it, the happier you are (usually). For LWC developers working on mobile apps, the editor of choice is VS Code. Salesforce provides multiple extensions for VS Code to make your mobile dev work easier, including Salesforce DX extensions, mobile extensions, and code validation.

- Visual Studio Code
- Salesforce DX
- Salesforce Extensions for VS Code
- Salesforce Mobile Extensions
- Code Validation

Preview

Get the current version of your code into a container that can run it. Previews need to be a fast, lightweight process, not a release ceremony. Salesforce mobile extensions for VS Code make this happen, whether you want to preview your components in development in a local preview environment, in the Test Harness app, or in an official Salesforce mobile app on a virtual or physical device.

- Preview Lightning Web Components on Mobile
- Salesforce Extensions for VS Code
- Salesforce Mobile Extensions
- Test Harness App

Test

Testing is an overloaded term in software development. There are many different kinds of testing. For your daily, interactive development work, the focus is on manually testing the behavior of the code you just added or changed. Once your component is in a preview environment, being able to tap on the app, navigate the user interface, and interactively play with your component or app is straightforward.

- Test Harness App
- Virtual Device Builds
- Salesforce Mobile App Betas

Debug

"Where did it all go wrong?" is a question every developer asks, usually many times a day. Debugging is how you answer that question.

"Why doesn't *X* happen when I do *Y*?" and "What *does* happen when I do *Y*?" are questions you need real debugging tools to help answer. LWC developers depend on standard debugging tools like Chrome DevTools and Safari Web Inspector to look inside their components to understand behavior. Mobile developers can use these same tools but, because LWC Offline code runs inside a mobile app instead of a web browser, it's a bit trickier to attach a JavaScript console to the web view. The mobile Test Harness app gives you not only a debuggable web view, but also a collection of specialized tools and functions to inspect the underlying behavior of your component code while running offline.

- Debug Lightning Web Components
- Virtual Device Builds
- Test Harness App
- Chrome DevTools
- Safari Web Inspector

Set Up Your Development Environment

Before you can create your first Lightning web component (LWC), or test a LWC in a mobile app, you must set up your development environment for mobile components.

The complete development environment for building Lightning web components for offline-enabled apps includes the following elements.

- Salesforce CLI (SFDX)
- Visual Studio Code (VS Code)

- Salesforce Extensions for VS Code
- Android Studio and a virtual device emulator
- Apple Xcode and a virtual device simulator
- Salesforce Mobile Extensions
- A virtual device build for the relevant mobile app

While you can eventually choose to replace or add tools to this list, start with these and add to them as you become more experienced.

Note: Don't let Android Studio or Xcode intimidate you. You only need to use it to provision a device emulator (Android) or simulator (iOS). You don't *need* both Android Studio and Xcode. If you want both, great. If not, pick the one you're most comfortable with.

Installation Instructions

We recommend the following installation sequence. Each link provides complete details for installing and configuring each of the different tools.

- Install and configure Android Studio
- Install and configure Xcode
- Install the Salesforce CLI
- Install the Mobile Extensions
- Install VS Code
- Install Salesforce Extensions for VS Code
- Download the latest virtual device build for the mobile app
  - For the Salesforce mobile app, use the standard virtual device builds, as described in Preview Components in the Salesforce Mobile App.

    Note: Your org must be enabled for LWC Offline before you can see or use LWC Offline features.

  - For the Field Service mobile app, the download link is posted in the Trailblazer Community.
- Install a virtual device build into a configured Android emulator or iOS simulator instance

The Develop a Lightning Web Component Quick Action quick start provides hands-on instructions for setting up your development environment, and also for creating, deploying, and running your first LWC.

IN THIS SECTION:

Set Up Xcode

Before you run previews in iOS simulators, make sure that Xcode 11 is properly installed and configured. After you install Xcode, test your environment with the Mobile Preview `setup` command. If you're using an existing Xcode installation, run the `setup` command to verify that your installed environment meets Mobile Extensions requirements.

Set Up Android Studio

Before you run previews in Android emulators, make sure that Android Studio is properly installed and configured. After you install Android Studio, test your environment with the `setup` command. If you're using an existing Android Studio installation, run `setup` to verify that your installed environment meets Mobile Extensions requirements.

[Install Mobile Extensions](#)

To use Mobile Extensions, install the `lwc-dev-mobile` Salesforce CLI plug-in. After it's installed, use it to check for the required Android and iOS configurations. If the plug-in finds problems, the command output gives you hints for how to fix your environment. After your environment is set up, preview your components from the command line or from Visual Studio Code.

Set Up Xcode

Before you run previews in iOS simulators, make sure that Xcode 11 is properly installed and configured. After you install Xcode, test your environment with the Mobile Preview `setup` command. If you're using an existing Xcode installation, run the `setup` command to verify that your installed environment meets Mobile Extensions requirements.

Install Xcode

How you install Xcode depends on your version of macOS. Mobile Extensions supports macOS version 10.14.4 (Mojave) and higher.

If you've already installed Xcode, skip to "Verify Your iOS Setup."

macOS Catalina, version 10.15.x

- If you're new to Xcode or are running Xcode 10 or earlier, install the latest version of Xcode from the Mac App Store . The latest version of Xcode meets the requirements of Mobile Extensions.

macOS Mojave, version 10.14.4 minimum

- Download and install Xcode 11.3.x from developer.apple.com/downloads/more. This site requires you to log in with your Apple ID.

Verify Your iOS Setup

To verify that your new or existing Xcode environment is ready for iOS previews, run the Mobile Extensions `setup` command:

```
sf force:lightning:local:setup -p ios
```

If `setup` reports any issues, use the following steps to correct them.

1. **Use a Mac:** Mobile Extensions for iOS requires a Mac running macOS Mojave version 10.14.4 and higher.
   a. On your Mac, click **About This Mac** in the System menu.
   b. Under **Overview**, look for the operating system name and version. Examples: "macOS Catalina Version 10.15.5", "macOS Mojave Version 10.14.4".
2. **Verify that you're running Xcode 11 or later.**
   a. In Xcode, select **Xcode** > **About Xcode**.
   b. Look for "Version 11.*x*.*y*" where *x* and *y* can be any integer values.
3. **Add an iOS simulator that runs iOS 13.**
   a. In **Xcode** > **Preferences**, select **Components**.

      In the Simulator list, installed simulators are marked with a blue check. The default simulator for your version of Xcode doesn't appear in this list.
   b. If none of your installed simulators use iOS 13, check and install at least one iOS 13 simulator.

4. **Final check:** Rerun the `setup` command:

```
sf force:lightning:local:setup -p ios
```

5. To see the list of installed virtual devices that Mobile Extensions recognizes on your machine, use the `device:list` command:

```
sf force:lightning:local:device:list -p ios
```

Set Up Android Studio

Before you run previews in Android emulators, make sure that Android Studio is properly installed and configured. After you install Android Studio, test your environment with the `setup` command. If you're using an existing Android Studio installation, run `setup` to verify that your installed environment meets Mobile Extensions requirements.

Install Android Studio

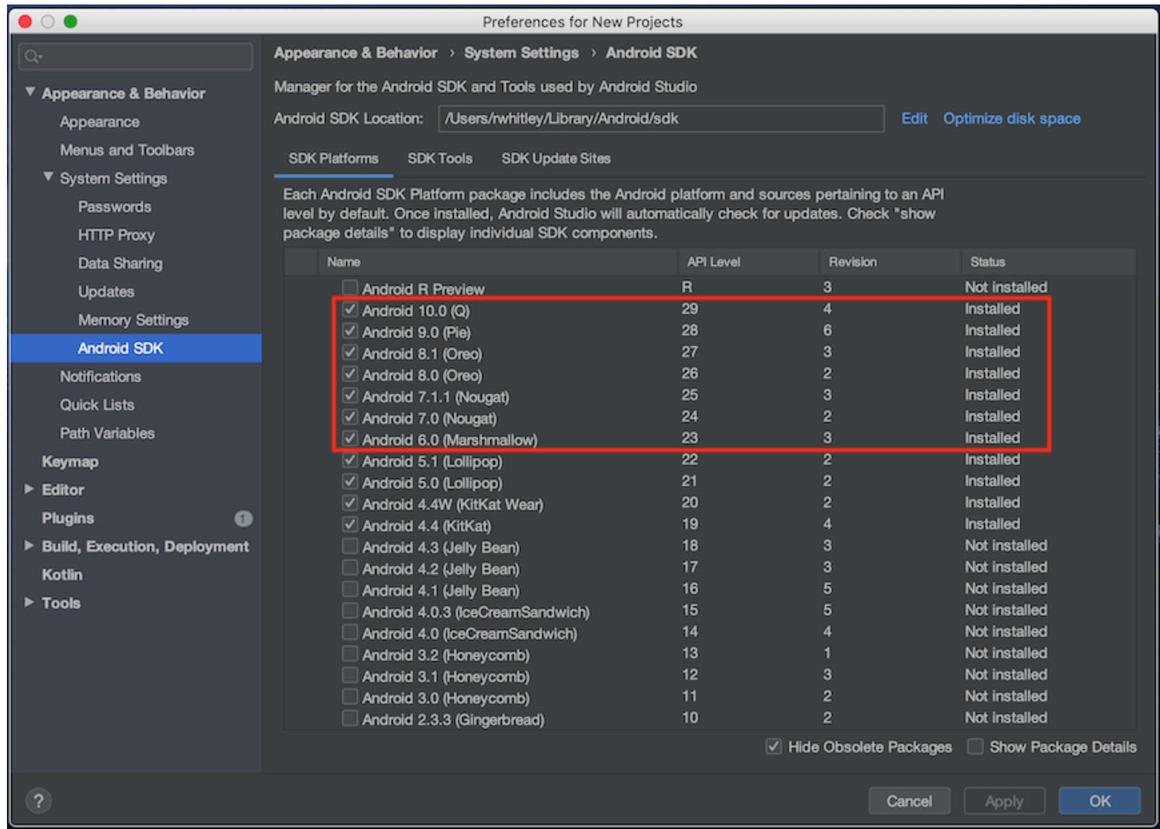
If you've already installed Android Studio, skip to "Verify Your Android Studio Setup."

To install Android Studio, download and run the installer at developer.android.com/studio.

Android apps specify different API levels. According to the Android documentation:

- "Minimum API level" is the lowest API level with which your app is compatible
- "Target API level" is the highest API level against which you've designed and tested your app

For previews, recommended API levels are API 23 (Android 6.0 "Marshmallow") to the latest available level. The Android Studio installer downloads only the latest API level. For Salesforce previews, you're free to add any other APIs from level 23 or later.

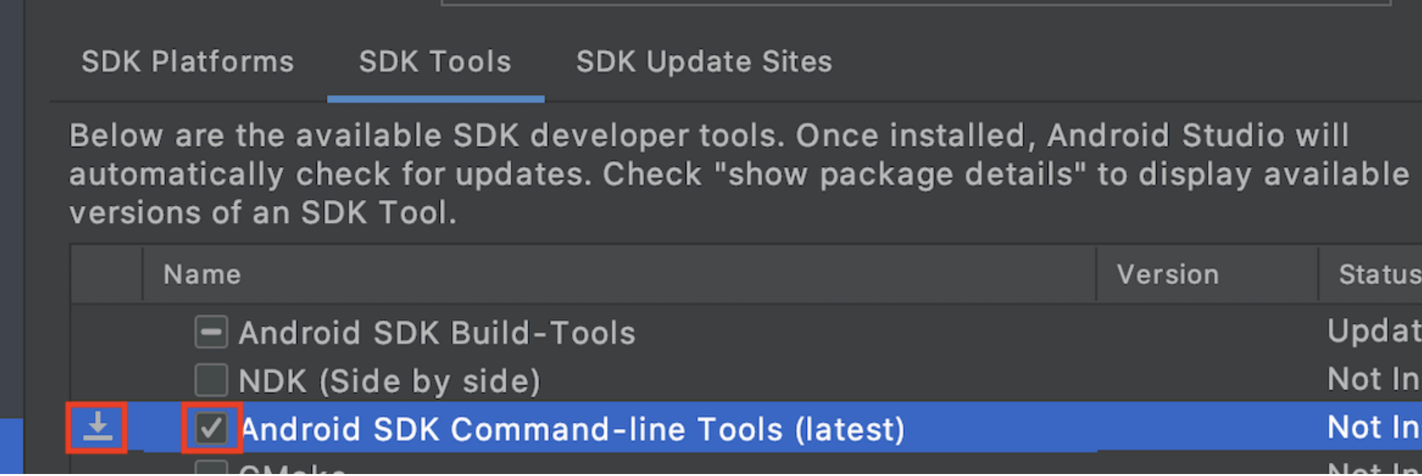


1. In the Android Studio Welcome screen, click **Configure** > **SDK Manager**.
2. If you require previewing with additional API levels, download one or more **SDK Platforms** from API level 23 or later. For each required level:
   a. Check the box next to its name.

   b. On the same row, click Download .

      Downloading new API versions can take several minutes.

   c. If prompted, confirm each download, then accept the license agreement and click **Next**.
3. Select **SDK Tools**.
4. If the status of **Android SDK Command-line Tools (latest)** is "Not installed", select **Android SDK Command-line Tools (latest)**, then click **Download**.



Although the latest version is recommended, any version of the command-line tools is expected to work.

5. When the downloads are finished, dismiss the SDK Manager.

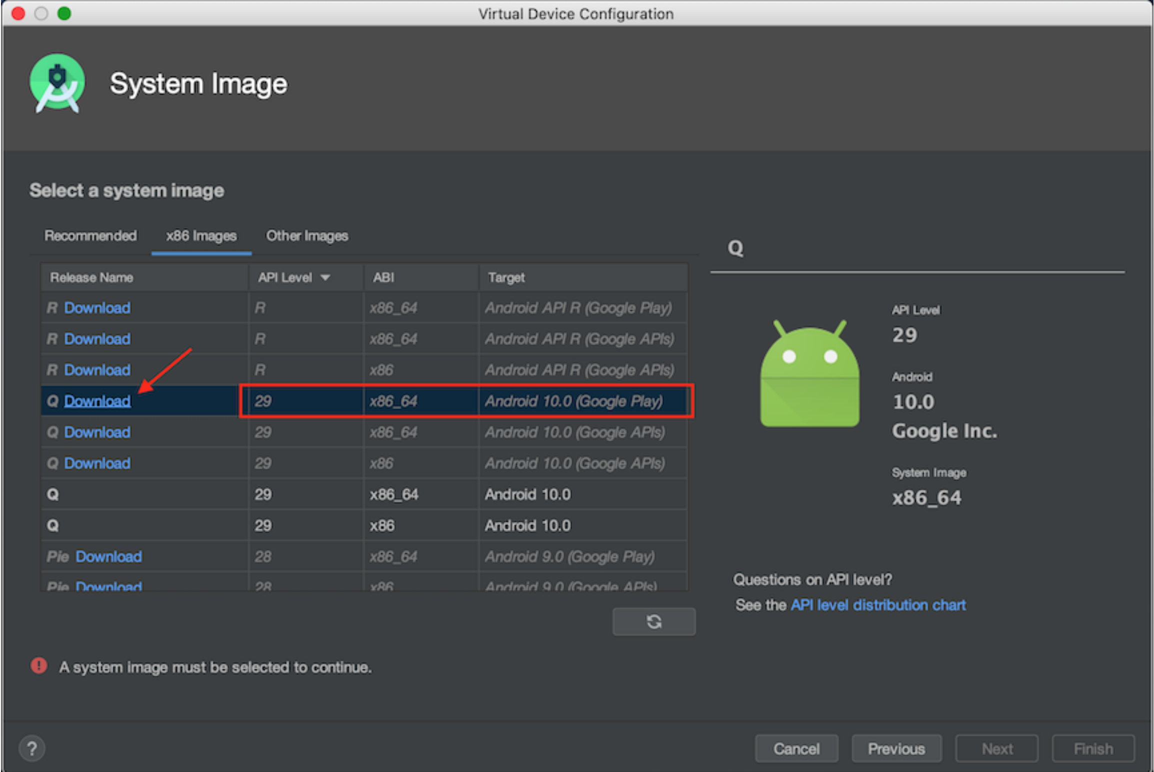
Create a Virtual Device

Now that you've installed the SDK, create an emulator for testing and debugging your apps. Let's choose a system image that supports API 23 or higher and Google APIs.

1. In the Android Studio Welcome screen, click **Configure** > **AVD Manager**.
2. On the Your Virtual Devices page, click **Create Virtual Device...**
3. Select a device definition, and click **Next**.
4. Under Select a System Image, click **x86 Images**.

   Note: In a new Android Studio installation, none of the images are downloaded.

5. Select an image listing that supports the following criteria.
   - **API Level:** 23 or higher
   - **ABI:** x86_64
   - **Target:** Any image that Android Studio makes available by default



6. Click **Download**.
7. After the download completes and you return to the System Image list, select the downloaded image and click **Next**.
8. In Verify Configuration, you can change the AVD name to any value that helps you identify the configuration. For other settings, you can accept the default values.
9. Click **Finish** to return to Your Virtual Devices.
10. To launch the emulator, click the Play button in your emulator's listing.

Verify Your Android Studio Setup

During installation, Android Studio configures your system environment with standard Android paths and locations. The Mobile Extensions `setup` command checks your settings and reports issues to the command-line console.

To verify that your environment is ready for Android previews, run the `setup` command:

```
sf force:lightning:local:setup -p android
```

If `setup` reports any issues, use the following guidelines to correct them.

System Variables

The `ANDROID_HOME` variable is the starting point for all Android development operations. If this variable isn't set or is incorrect, Mobile Extensions can't use Android tools. You set `ANDROID_HOME` to point to the top-level directory of your Android SDK installation. On macOS, for example, this variable's default value is `/Users/<user_name>/Library/Android/sdk`. If you've installed Android SDK in a custom directory, make sure that this path points to that location.

macOS X:

You can make these settings persistent by adding them to the `~/.bash_profile` or similar startup file for your command shell.

- Configure `ANDROID_HOME` as follows.

```
export ANDROID_HOME=/Users/<your_user_name>/Library/Android/sdk
```

- For Android, the Mobile Extensions plug-in requires the Java Development Kit (JDK) version used by Android Studio. Currently, this version is JDK 8. Your plug-in commands might fail if:
  - A `JAVA_HOME` system variable exists on your machine and points to a different JDK version.
  - Other applications in your system path use a different JDK version.

  You can find the correct JDK version installed with Android Studio's Java runtime engine embedded in the Android Studio app. For example:

```
export JAVA_HOME="/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home"
```

  See the Android Developer documentation.

Windows: In **System Properties** > **Advanced**, add the following variable definitions to either System Environment Variables or User Environment Variables. (For help with finding System Properties, see www.imatest.com/docs/editing-system-environment-variables.)

- - Variable: `ANDROID_HOME`
  - Value: `C:\Users\<user>\AppData\Local\Android\Sdk`
- - Variable: `JAVA_HOME`
  - Value: `C:\Program Files\Android\Android Studio\jre`

  Note: For Android, the Mobile Extensions plug-in requires the Java Development Kit (JDK) version used by Android Studio. Currently, this version is JDK 8. If you set JAVA_HOME to a custom path that points to a different JDK version, the plug-in commands might fail.

Linux: Set the following variable definitions in your `~/.profile` file.

- - Variable: `ANDROID_HOME`
  - Value: `~/Android/Sdk`
- - Variable: `JAVA_HOME`
  - Value: `/snap/android-studio/current/android-studio/jre`

  Note: For Android, the Mobile Extensions plug-in requires the Java Development Kit (JDK) version used by Android Studio. Currently, this version is JDK 8. If you set JAVA_HOME to a custom path that points to a different JDK version, the plug-in commands might fail.

Android SDK Version

One or more of: Android 6.0 (Marshmallow), API Level 23, or higher.

Android Tools and Emulator

For the following items, verify that you've installed the indicated version.

- Android Emulator: 23 or higher
- Android SDK Platform-Tools: 23 or higher
- Android SDK Command-line Tools: Any version (recommended: Latest)

Check these versions in the SDK Tools section of the Android SDK Manager.

Note: In addition to the version ranges listed here, later stable versions of these tools are also expected to work.

If you've changed any settings, rerun the `setup` command.

Install Mobile Extensions

To use Mobile Extensions, install the `lwc-dev-mobile` Salesforce CLI plug-in. After it's installed, use it to check for the required Android and iOS configurations. If the plug-in finds problems, the command output gives you hints for how to fix your environment. After your environment is set up, preview your components from the command line or from Visual Studio Code.

The Mobile Extensions plug-in provides several tools:

Setup
Run `force:lightning:local:setup` to set up virtual mobile devices—iOS simulators and Android emulators—in their local environments. Mobile Extensions provides a preconfigured default virtual device for iOS and for Android.

Preview
Run `force:lightning:lwc:preview` to preview your components on virtual mobile devices. You can choose either the default device or one that you've configured. Mobile Extensions let you add and recall virtual device configurations. The plug-in presents your component preview in the default browser of your virtual device. This preview acts as a playground—instantly reflecting visual changes you apply to your component's code.

List Devices
Run `force:lightning:local:device:list` to see a list of virtual devices found on your machine by the Mobile Extensions plug-in.

Install the Mobile Extensions Plug-in

1. If you haven't yet installed Salesforce CLI on your development machine, see Install the Salesforce CLI.
2. Install the Mobile Extensions plug-in:

```
sf plugins install @salesforce/lwc-dev-mobile
```

   A message that the plug-in isn't digitally signed is expected and isn't a cause for concern.
3. If you're prompted to continue installation, enter *y*.
4. To see the available commands, use `grep` with `sf commands`, which lists the commands with a space separator rather than a colon. Salesforce CLI commands accept either separator. For example, the `setup` command is in the `force lightning local` topic, while the `preview` command is in the `force lightning lwc` topic:

```
sf commands | grep "force lightning local"
 force lightning local device create          Create a virtual mobile device


 force lightning local device list           List the available virtual mobile devices
 for Lightning Web Component development.

 force lightning local device start           Start a virtual mobile device


 force lightning local setup                  Set up mobile environment for Lightning
Web Component development.
```

```
sf commands | grep "force lightning lwc"
 force lightning lwc preview                      Preview Lightning Web Components in a
virtual mobile environment

 force lightning lwc start                        Develop Lightning Web Component modules
and see live changes without publishing your components to an org.

 force lightning lwc test create                  creates a Lightning web component test
file with boilerplate code inside a __tests__ directory.

 force lightning lwc test run                   invokes Lightning Web Components Jest unit
 tests.

 force lightning lwc test setup                   install Jest unit testing tools for
Lightning Web Components.

 force lightning lwc test ui mobile configure Create a configuration file for running
UTAM tests on mobile.

 force lightning lwc test ui mobile run       Run UTAM test by specifying a WDIO
configuration. Test specs to run can be explicitly specified by using a flag.
```

To see the syntax of the available commands, call each command with the `--help` argument. For example:

```
sf force:lightning:local:setup --help
sf force:lightning:lwc:preview --help
```

Check Your Development Environment

To verify that your development environment meets Mobile Extensions requirements, run the `setup` command. This command runs a series of tests that checks the presence and versions of your mobile platform libraries, tools, and virtual device configurations. If the command detects problems, it prints hints for solving them in the Terminal console or Windows command prompt.

You can run this command to check either an Android or an iOS configuration.

```
sf force:lightning:local:setup -p <platform_name>
```

- `-p`, `--platform`: Mobile platform to verify. Can be either `Android` or `iOS` (case insensitive).
- Two other parameters—`--json` and `--loglevel`—are standard Salesforce CLI conventions and aren't part of the `setup` functionality.

1. Check your Android setup status.

   ```
   sf force:lightning:local:setup -p android
   ```

   a. If your setup is correct, this command prints "Passed" status for all tests. You're ready to start previewing components on Android!
   b. If any test fails, follow the suggestions printed in the console to finish setting up your Android environment. To review setup instructions, see Set Up Android Studio on page 175.
   c. Repeat this step until all tests report "Passed".

2. Check your iOS setup status.

   ```
   sf force:lightning:local:setup -p ios
   ```

a. If your setup is correct, this command prints "Passed" status for all tests. You're ready to start previewing components on iOS!
b. If any test fails, follow the suggestions printed in the console to finish setting up your iOS environment. To review setup instructions, see Set Up Xcode on page 174.
c. Repeat this step until all tests report "Passed".

Check Your Available Virtual Devices

To see which of your devices are available to Mobile Extensions, use the following command.

```
sf force:lightning:local:device:list -p <platform_name>
```

- `-p`, `--platform`: Mobile platform to verify. Can be either `Android` or `iOS` (case insensitive).

Note:

- The preview command allows you to select any of your installed simulators. However, Salesforce can't guarantee preview performance on all virtual devices.
- For iOS, the list produced by this command uses the format "*device name*, *iOS_runtime version*".
- For Android, the list produced by this command uses the format "*device name*, *device type*, *Android runtime version*".
- For Android, if your development environment variables aren't properly configured, this command returns an empty list.

Create a Virtual Device

To create an iOS or Android device that's guaranteed to meet Mobile Extensions requirements, use the following command:

```
sf force:lightning:local:device:create -n <name> -d <device_type> -p [ios|android]
    [-l <android_api_level>]
```

- **`-n, --devicename`**

  Name for the new virtual device. If the specified name is already in use on your machine, the command fails with an explanatory error. To see the list of your existing device names, use the `force:lightning:local:device:list` command.

  `-d, --devicetype`

  Type of virtual device.

  If the command doesn't recognize your device type entry, choose one from the list provided by the resulting error message. For example:

  - Android:

  ```
  □ FAILED: Validating specified device type. (0.001 sec)
     › Device type 'pixel_xlst' is invalid. Must be one of the following
       valid types: pixel, pixel_xl, pixel_c
  ```

  - iOS:

  ```
  □ FAILED: Validating specified device type. (0.287 sec)
     › Device type '"iPhone 12 Pro Max 14.4"' is invalid. Must be one of the following

       valid types: iPhone-8, iPhone-8-Plus, iPhone-X, iPhone-XS,
       iPhone-XS-Max, iPhone-XR, iPhone-11, iPhone-11-Pro, iPhone-11-Pro-Max,
       iPhone-12-mini, iPhone-12, iPhone-12-Pro, iPhone-12-Pro-Max
  ```

-p, --platform
Mobile platform to create. Can be `Android` or `iOS` (case insensitive).

-l, --apilevel
(Optional, Android only) Android API level for the new device. Default value is your latest installed API level. Not used for iOS devices.

- Two other parameters—`--json` and `--loglevel`—are standard Salesforce CLI conventions and aren't part of the `setup` functionality.

iOS Example

```
sf force:lightning:local:device:create -p ios -n 'iOS test' -d 'iPhone 12 Pro Max'
```

Android Example

```
sf force:lightning:local:device:create -p android -n 'API 29 pixelXL' -d 'pixel_xl'
-l 29
```

This command starts by running the `force:lightning:local:setup` command. If that command reports failures, address the first reported failure, and then retry the `force:create` command.

If you specify a device name that already exists on your machine, the command exits with the message "A virtual device with the name '<*existing_name*>' already exists."

Start a Virtual Device

To launch a virtual device, use the following command.

```
sf force:lightning:local:device:start -p [ios|android] -t <virtual_device_name>]
    [-w <true|false>]
```

- **-p, --platform**
  Mobile platform of the target device. Can be `Android` or `iOS` (case insensitive).

  -t, --target
  Name of the virtual device to launch.

  -w, --writablesystem
  (Optional, Android only) Doesn't accept a value. If present, the virtual device launches with a writable system. Otherwise, the system is read-only.

- Two other parameters—`--json` and `--loglevel`—are standard Salesforce CLI conventions and aren't part of the `setup` functionality.

iOS Example

```
sf force:lightning:local:device:start -p ios -t "iPhone 12 Pro Max"
```

Android Examples

```
sf force:lightning:local:device:start -p android -t "API 29 pixelXL"
```

```
sf force:lightning:local:device:start -p android -t "API 29 pixelXL" -w
```

Preview Lightning Web Components on Mobile

When you're developing Lightning web components, it's important to inspect your components' presentation not only on the desktop, but also on mobile devices. To preview your components on virtual mobile devices and see changes as you code, use the Salesforce CLI Mobile Extensions plug-in. Then download and run virtual device builds of the Salesforce mobile app to preview how your components coexist with other components in Salesforce.

You can provide feedback and suggestions for Mobile Extensions and Salesforce mobile app downloadable builds in the Mobile Tools Trailblazer Community.

IN THIS SECTION:

Mobile Development Preview Environments

While coding components, developers can launch mobile previews from VS Code or from the command line. Developers can also preview components in context in the Salesforce mobile app, or in a virtual device build of the Salesforce mobile app. Salesforce admins can preview Lightning Experience on mobile from Lightning App Builder.

Preview Components from the Command Line

After you've installed the Mobile Extensions plug-in and set up Xcode and Android Studio, you can launch previews directly from the command line. As you code, mobile previews immediately reflect your changes.

Preview Components from Visual Studio Code

After you've installed the Mobile Extensions plug-in and set up Xcode and Android Studio, install Salesforce Extensions for Visual Studio Code. Now you can launch mobile previews from the VS Code command palette. Mobile previews immediately reflect visual changes you make to your component as you edit.

Preview Components in the Salesforce Mobile App

To verify your Lightning web components in Salesforce on many devices, use virtual device builds of the Salesforce mobile app. These builds make it possible to run Salesforce on iOS simulators and Android emulators.

Preview Components in Custom Mobile Apps

If you develop your own custom native apps for iOS and Android, you can adapt them to preview Lightning web components. You provide configuration for installing your app on mobile devices and implement some means of hosting the preview. At runtime, you use the advanced features of the `lwc:preview` command to send previews to your app.

Mobile Development Preview Environments

While coding components, developers can launch mobile previews from VS Code or from the command line. Developers can also preview components in context in the Salesforce mobile app, or in a virtual device build of the Salesforce mobile app. Salesforce admins can preview Lightning Experience on mobile from Lightning App Builder.

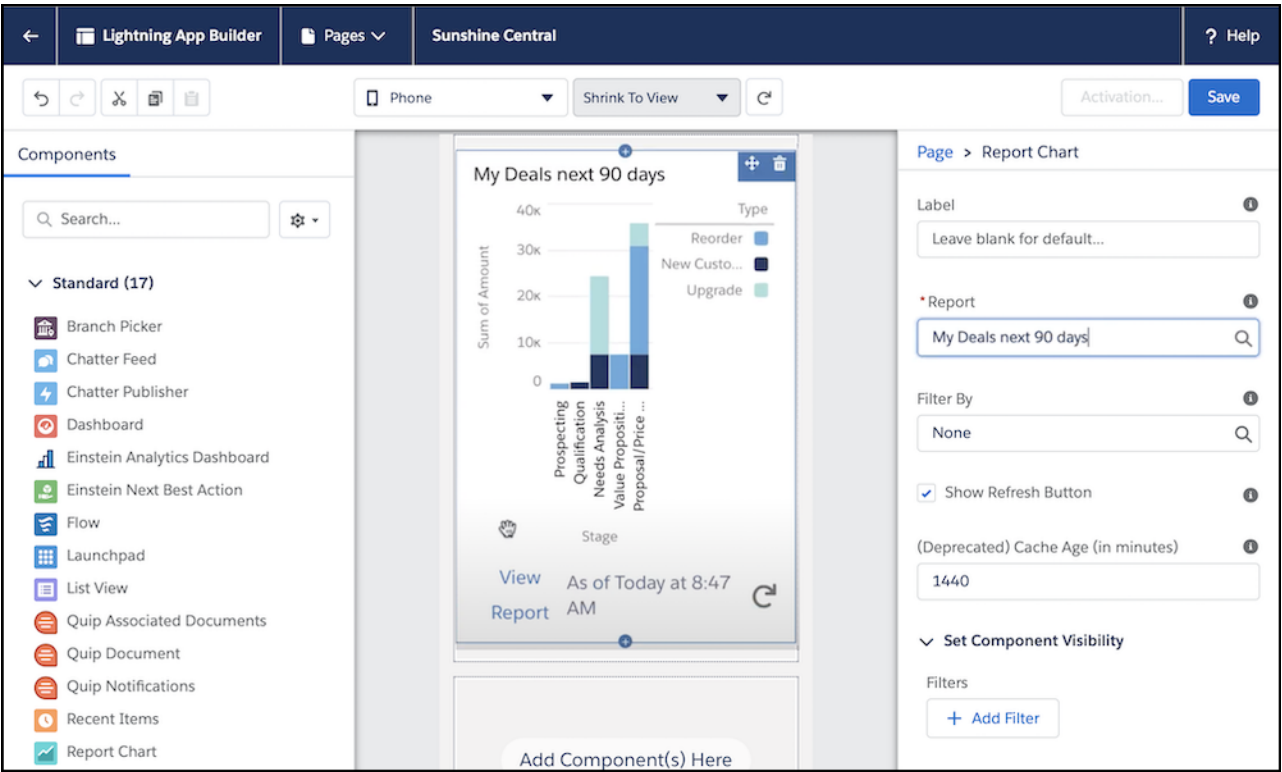
Administrators

Salesforce Administrators work with stakeholders to define requirements and to customize their org, using the full variety of tools available in Salesforce. When customizing pages, administrators should validate their changes for both desktop and mobile.

App Builder Mobile Previews

Admins can use Lightning App Builder to build apps and pages to customize Lightning Experience on mobile. App Builder provides previews for both desktop and mobile. Admins can specify certain components as mobile- or desktop-only. See Visibility Rules on Lightning Pages for available filtering options.

For more information about building and previewing pages in App Builder, see Build a Mobile App Page and Get Ready for Lightning — 5 step-by-step videos.



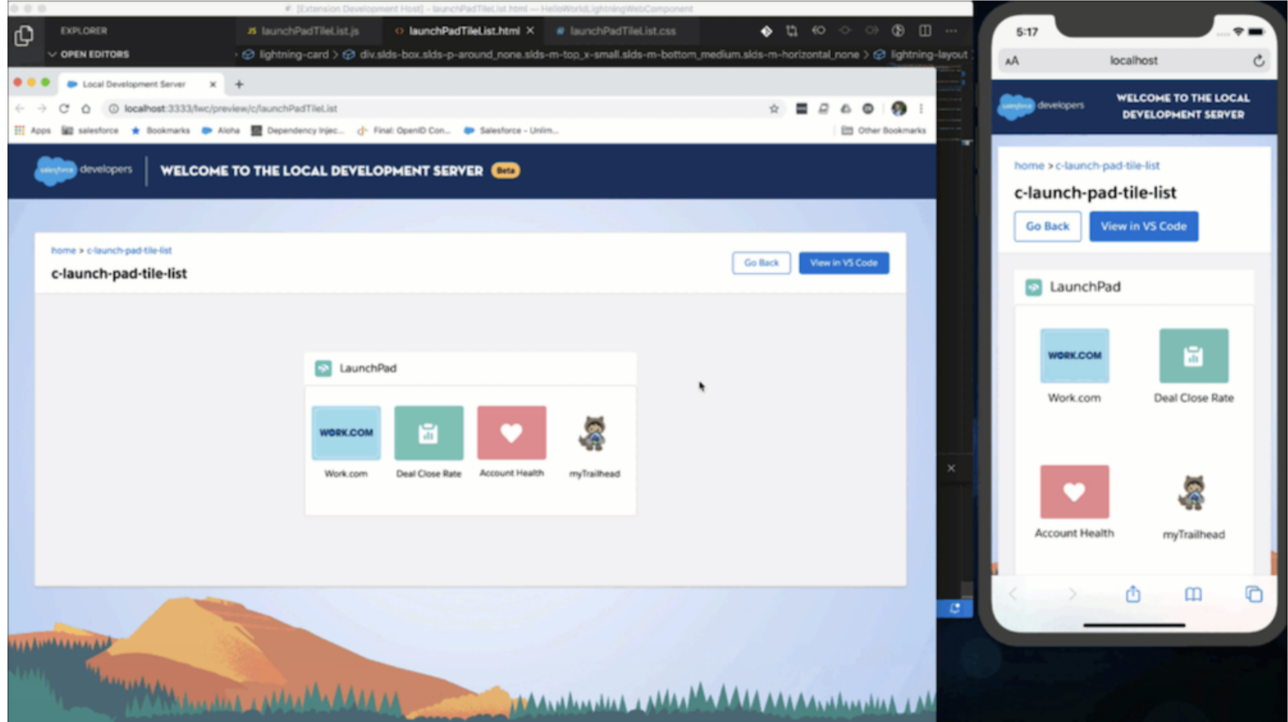
If admins need more than a preview, they can run Lightning applications and components in the Salesforce mobile app on their devices. Testing in the mobile app helps validate that users have the best experience on mobile.

Developers

Salesforce developers building Lightning web components have many options available to validate their components for mobile.

Preview with Local Development

During development you can view mobile previews for Lightning web components locally with the Salesforce Extensions for Visual Studio Code. You can launch a mobile preview from the Command Palette while you're editing a component. Mobile previews immediately reflect visual changes you make to your component as you edit, before publishing to Salesforce. Local previews can display your component in a web browser on your desktop and on a simulated mobile device.



Learn more about mobile previews using local development tools.

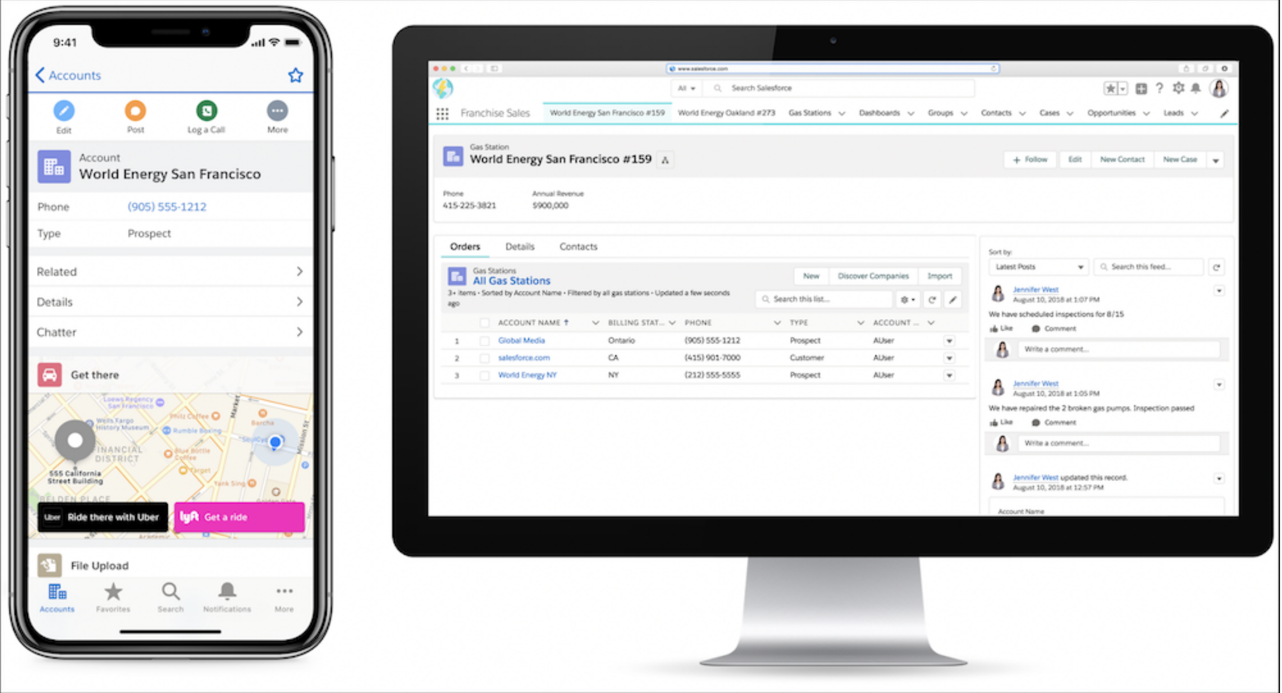
- Preview Components from the Command Line on page 188
- Preview Components from Visual Studio Code on page 191
- Local Development (Beta)

Preview in the Salesforce Mobile App

Previews using the local development server are great for immediately seeing the effects of changes. After your component is validated locally, publish it to a Salesforce development org and validate it on pages with other components in the Salesforce mobile app. You can run the Salesforce mobile app on physical devices and on virtual devices.

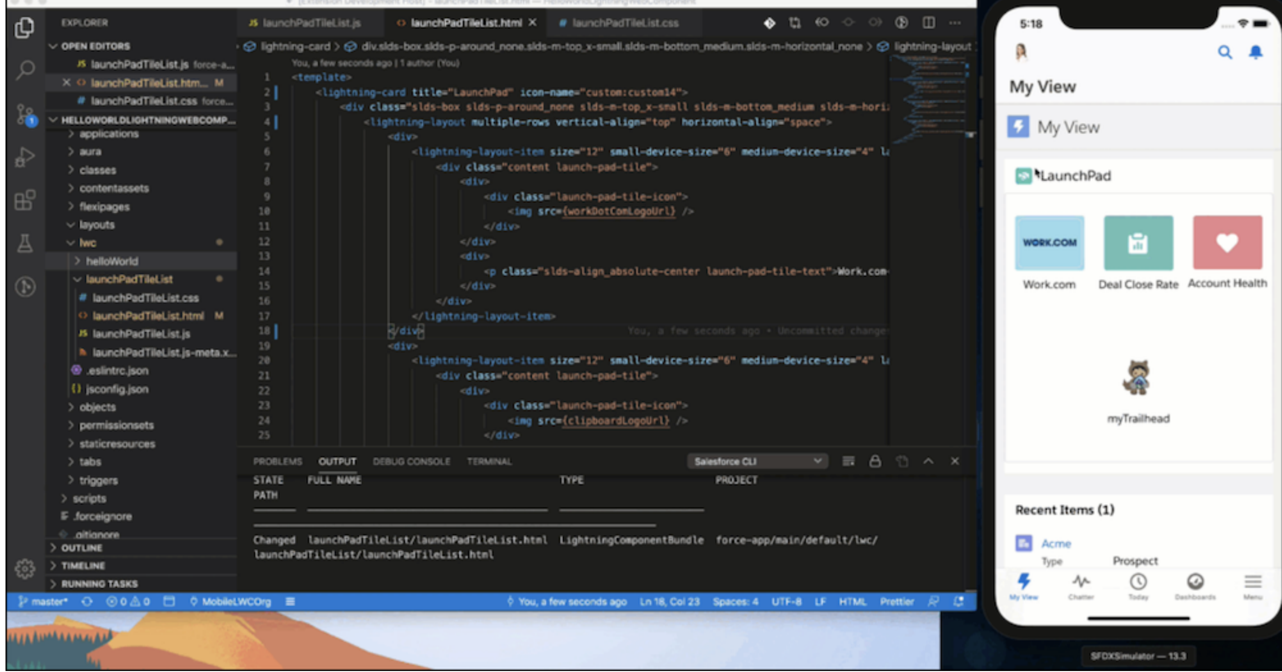
- **Physical Devices**

  You can run your Lightning applications and components using the Salesforce mobile app on your device. Use the mobile app to review issues your applications and components encounter on mobile. The Salesforce mobile app is available for download from the App Store and Google Play.



- **Virtual Device Builds**

  You can run your Lightning applications and components using virtual device builds of the Salesforce mobile app. These builds make it possible to run Salesforce on iOS simulators and Android emulators so you can validate and debug your components on many devices. To learn how to download virtual device builds, see Preview Components in the Salesforce Mobile App on page 192.



Preview Components from the Command Line

After you've installed the Mobile Extensions plug-in and set up Xcode and Android Studio, you can launch previews directly from the command line. As you code, mobile previews immediately reflect your changes.

The `force:lightning:lwc:preview` command supports two sets of parameters: three basic parameters and four advanced parameters. Every call requires the first two basic parameters and, optionally, the third. Advanced parameters let you designate and configure a custom app to host the preview.

```
sf force:lightning:lwc:preview -p <platform_name> -n <component_name>
    [-t <target_virtual_device_name>] [-a browser|<app_identifier>]
    [-d <project_dir>] [-f <preview_config_file>] [--confighelp]
```

Basic Parameters

- `-n`, `--componentname`:

  Name of your component.

- `-p`, `--platform`:

  Mobile platform to use for the preview. Can be either `Android` or `iOS` (not case sensitive).

- (Optional) `-t`, `--target`:

  Name of a target virtual device. This device is the iOS simulator or Android emulator configuration that hosts the preview. You can pass the name of a new or an existing device. If you enter a name that's not recognized by the selected platform, this command creates the device using the system default configuration. If omitted, the command launches the default virtual device for the given platform. See the **Managing Devices** section.

Advanced Parameters

- (Optional) `-a`, `--targetapp`

  Target app for the preview. Acceptable values are `browser`, which means the default mobile browser, or an app ID. Defaults to `browser`.

- (Optional) `-d`, `--projectdir`

  Root directory of the CLI project. Defaults to the current working directory.

- (Optional) `-f`, `--configfile`

  File named `mobile-apps.json` that specifies extended configuration options for the preview. If you pass an app ID to `--targetapp`, you're required to provide this file. Use the `-f` parameter to specify a custom path to this file. This path can be relative (to the project's working directory) or absolute.

- (Optional) `--confighelp`

  Displays the schema of the extended configuration file.

Here's a macOS example of a basic call. The name of your component is `helloWorld`, and you're developing it in the `~/Projects/helloWorld/` directory.

1. In a Terminal window, `cd` to the directory of your Lightning Web Components project.

   ```
   cd ~/Projects/helloWorld/
   ```

2. Start the Lightning Web Components server.

   ```
   sf force:lightning:lwc:start
   ```

Because the server is a synchronous process, this window doesn't accept further command input until the server is stopped.

3. Leaving the server running, open a second Terminal window or Windows command prompt in your project directory, and enter this command:

```
sf force:lightning:lwc:preview -p Android -n HelloWorld -t "Pixel XL API 29"
```

Launching the virtual device can take a few seconds. After it has booted, the Mobile Extensions plug-in presents your component in the device's default browser. You can inspect and interact with your component. If you change visual aspects of your component's code, the simulator immediately reflects those changes without requiring a manual refresh.

Managing Devices

To manage devices:

- **iOS:** Use the Devices and Simulators tool in Xcode.
- **Android:** Use the Android Virtual Device (AVD) Manager.

To see the list of supported virtual devices in your environment, use the `device:list` command:

- **iOS:**

```
sf force:lightning:local:device:list ios
```

- **Android:**

```
sf force:lightning:local:device:list android
```

Use this command only after your development environment is fully configured and operational. Virtual device images are required to match or exceed system requirements for Mobile Extensions. In addition, the following scopes can limit the number of devices shown in the list.

iOS:
Devices must run iOS or iPadOS.

Android:
No known limitations.

Here are a few tips on how the device list works.

- If you enter a name that doesn't match an existing virtual device, Mobile Extensions creates a device with that name.
- Android virtual devices created by Mobile Extensions are based on the latest SDK level and use Google APIs.

Configuring a Native Mobile App to Host Previews

You can use the `force:lightning:lwc:preview` command to launch a preview in an iOS or Android native mobile app that you provide. No Salesforce-specific requirements apply to these custom native apps. Your app simply must provide some means of displaying Lightning web components.

To launch previews in your custom app, the Mobile Extensions plug-in requires information about the app's identity and its access points. You specify the following properties in the JSON configuration file you specified in the `--configfile` parameter.

`id`

iOS

The app's ID in reverse-DNS format.

Android

The app's ID in the format supported by the Android Debug Bridge (`adb`) shell. For example, here's the format that the adb `start` command accepts:

```
adb shell am start -n com.package.name/com.package.name.ActivityName
```

`name`

The app's "friendly" name. Salesforce VS Code Extensions use this name to identify the app.

`get_app_bundle` (Optional)

Name of a Node.js or TypeScript module that provides configuration for installing the app on the target device or emulator.

This module must expose a `run()` method that returns the path to the app bundle as a `String`. If this path is relative rather than absolute, Mobile Extensions calculates the path relative to the configuration file. This module is also responsible for providing any implementation details that allow access to the app bundle.

If the `get_app_bundle` argument is present, Mobile Extensions calls the `run` command before launching the preview. If omitted, Mobile Extensions assumes that the bundle is already installed on the target device or emulator.

`activity` (Android only)

The class name of the app's main activity in a format that supports launching the app from the Android Debug Bridge (`adb`) shell. For example, here's the format that the adb `start` command accepts:

```
adb shell am start -n com.package.name/com.package.name.ActivityName
```

In your configuration file, you use `.ActivityName` (including the leading period.)

`launch_arguments` (Optional)

An array of objects specifying arguments required for launching the app. Each object contains one argument's name and value. Each argument name and argument value is expressed as a name-value pair.

`preview_server_enabled` (Optional)

Indicates whether the native app requires a local development server for previewing a component. If false (default value), the preview command proceeds without checking for a running server instance. If true, before beginning the preview the command checks for a running server instance. If this check fails, the command returns an error and prompts the user to start the server before rerunning the command.

Properties other than `launch_arguments` are simple name-value string pairs. Here's a sample configuration file.

```
{
  "apps": {
    "ios": [
      {
        "id": "com.salesforce.mobiletooling.lwctestapp",
        "name": "LWC Test App",
        "get_app_bundle": "configure_ios_test_app.ts",
        "launch_arguments": [
          { "name": "ShowDebugInfoToggleButton", "value": "true" },
          { "name": "username", "value": "Astro" }
        ],
        "preview_server_enabled": true
      }
    ],
    "android": [
      {
        "id": "com.salesforce.mobiletooling.lwctestapp",
        "name": "LWC Test App",
```

```
        "get_app_bundle": "configure_android_test_app.ts",
        "activity": ".MainActivity",
        "launch_arguments": [
          { "name": "ShowDebugInfoToggleButton", "value": "true" },
          { "name": "username", "value": "Astro" }
        ],
        "preview_server_enabled": true
      }
    ]
  }
}
```

You can access the schema for this file at /src/cli/commands/force/lightning/lwc/previewConfigurationSchema.json in the `forcedotcom/lwc-dev-mobile` GitHub repo.

SEE ALSO:

Preview Components in Custom Mobile Apps

Preview Components from Visual Studio Code

After you've installed the Mobile Extensions plug-in and set up Xcode and Android Studio, install Salesforce Extensions for Visual Studio Code. Now you can launch mobile previews from the VS Code command palette. Mobile previews immediately reflect visual changes you make to your component as you edit.

When you launch a mobile preview, the extension initializes a virtual device for the platform you selected—an Android emulator or an iOS simulator. You can choose:

- The default virtual device
- A compatible device that you've configured in Android Studio or Xcode
- A new named device based on the default configuration, using a custom name that you provide
- Your most recently used named device, if one exists

1. To install the Visual Studio Code extension, follow the instructions for Salesforce Extensions for Visual Studio Code.
2. While in Settings, you can also customize extension features.
   a. For **Log Level**, select the degree of granularity that you prefer for compiler messages.
   b. Check **Remember Device** to easily recall the named device that you most recently used.
3. In Visual Studio Code, open your Lightning Web Component project.
4. To preview your component, use one of the following methods.
   - In your project, right-click your component folder and select **SFDX: Preview Component Locally**.
   - In the Command Palette, enter *`preview component`*, and select **SFDX: Preview Component Locally**.
5. Select **Use Android Emulator** or **Use iOS Simulator**.

   The extension searches your computer for virtual devices on the selected platform. It then displays a list of all discovered virtual device names that support Salesforce mobile previews.
6. Do one of the following:
   - If a list of remembered device names appears, select a device from the list.
   - Enter the name of a new virtual device.

- Leave the field blank to use the default configuration.

7. For previews in custom native apps only: Select whether to preview the component on your mobile browser or in a custom native app. Custom apps appear in the list when your Lightning web component project has mentioned them in its configuration file. For VS Code, this file is required to be in the root directory of your project.

SEE ALSO:

Run Local Development SFDX Commands in VS Code

Preview Components in the Salesforce Mobile App

To verify your Lightning web components in Salesforce on many devices, use virtual device builds of the Salesforce mobile app. These builds make it possible to run Salesforce on iOS simulators and Android emulators.

You can always inspect your components in Salesforce on physical mobile devices. However, if you're testing a great variety of models, you could quickly deplete your hardware budget. To test more efficiently, run Salesforce virtual device builds on virtual representations of iOS and Android devices.

Virtual device builds come in packages that you can drop onto a running instance of an iOS simulator or an Android emulator. To obtain simulators and emulators, install each platform's development environment: Xcode for iOS, and Android Studio for Android. You can run Salesforce on any supported simulator or emulator provided with these environments. The device you choose must be running iOS 12 or later, or Android 7.0 or later.

Salesforce has tested the Salesforce Android version on Google and Samsung devices, and the iOS version on iPhones and iPads. For a list of tested devices, see "Requirements for the Salesforce App" in Salesforce Help.

Tip: Salesforce mobile app isn't the only Salesforce app that supports virtual device builds. For example, you can also use the following instructions with Salesforce Field Service.

Install Xcode and Simulators

If you haven't installed Xcode 12, see Install Xcode on page 174. Salesforce mobile app requires Xcode 12 or later.

After you've installed Xcode 12, verify your system environment. See Verify Your iOS Setup on page 174.

Install Android Studio and Emulators

If you haven't installed Android Studio and Android emulators, see Install Android Studio on page 175.

After you've installed Android Studio, verify your system environment. See Verify Your Android Studio Setup on page 178.

Download iOS and Android Mobile App Packages

To obtain virtual device builds, you download the mobile app package files that contain them. For iOS, the package is a ZIP file. For Android, it's an APK file. The packages are publicly available, don't require you to log in, and don't automatically expire.

Note: To download packages for other apps (like Salesforce Field Service), substitute the package download URL and the iOS app file name as specified in the other app's documentation.

iOS

Package (ZIP) file download: sfdc.co/salesforce-mobile-app-ios-simulator

Extracted file name: Chatter.app

Android

Package (APK) file download: sfdc.co/salesforce-mobile-app-android-emulator

Install iOS and Android Virtual Device Builds

You can use Android builds on Mac and Windows operating systems. You can use iOS builds on Mac. Install the virtual device build on each simulator or emulator you intend to use.

iOS

1. In Finder, unzip the downloaded package file to extract the app file.
2. Drag the app file from Finder and drop it on a running iOS simulator instance.
3. On the simulator, find the app's icon and launch the app.

Android

1. Drag the downloaded APK file from Mac Finder or Windows Explorer and drop it on a running Android emulator instance.
2. On the emulator, find the app's icon and launch the app.

Install Your Component in a Salesforce Org

You can develop Lightning web components in a Salesforce CLI (SFDX) project. If you're new to SFDX projects for Lightning Web Components, get started quickly with this Trailhead project.

From VS Code, it's easy to install your component in a scratch org.

1. In VS Code, enter `Command-Shift-P` or `Ctrl-Shift-P` to launch the Command Palette.
2. If you don't have a scratch org:
   a. In the input field, enter `SFDX: Create a Default Scratch Org`, then click the matching VS Code listing.
   b. Follow the prompts.
   c. In the input field, enter `SFDX: Push Source to Default Scratch Org`, then click the matching VS Code listing.
   d. As always, watch Output in the console panel to verify that everything goes smoothly.

Verify Your Components

1. Log in to the org where your component is installed.
2. Inspect and verify your component's visibility and appearance in relation to other components.
3. Inspect the Salesforce web view by attaching Safari (iOS) or Chrome (Android) developer tools to the virtual device.

See Debug Your Components with Virtual Device Builds on page 216 for detailed instructions.

SEE ALSO:

Salesforce Help: Salesforce Mobile App

Trailhead: Salesforce Mobile App Customization

Preview Components in Custom Mobile Apps

If you develop your own custom native apps for iOS and Android, you can adapt them to preview Lightning web components. You provide configuration for installing your app on mobile devices and implement some means of hosting the preview. At runtime, you use the advanced features of the `lwc:preview` command to send previews to your app.

The best way to learn about custom app previews is to study the examples at github.com/forcedotcom/LWC-Mobile-Samples. This repo contains sample native mobile apps built to host previews, and sample Lightning Web Components projects for previewing in those apps. Binding these artifacts together are the advanced parameters of the `lwc:preview` command.

HelloWorld: A Sample Lightning Web Component Project

The HelloWorld project in the top level of the LWC-Mobile-Samples repo is the simplest place to start. This Lightning Web Components project satisfies the `lwc:preview` requirements by including a JSON configuration file and mobile app bundle scripts. You can find these files in the root folder of the HelloWorld project:

- JSON configuration file: `mobile-apps.json`
- iOS bundle script file: `configure_ios_test_app.ts`
- Android bundle script file: `configure_android_test_app.ts`

You specify the bundle script file names in the `get_app_bundle` pairs of `mobile-apps.json`.

Custom Native Sample Apps

In the `apps` folder of the LWC-Mobile-Samples repo, you can find the iOS and Android native apps that the HelloWorld component targets. These native apps are bare-bones projects that each add a container for Lightning web component previews to what's essentially a template app. In iOS, the container is an instance of `WKWebView`. In Android, it's a `WebView` object. These two classes—`WKWebView` and `WebView`—are implementation-dependent choices. For your own app, you can use whatever works best for hosting web content.

The iOS and Android apps follow the same preview flow:

1. Parse the configuration values and launch arguments from the JSON file.
2. Use the launch arguments to create a preview URL. If the `preview_server_enabled` argument is true but the launch arguments don't include a web domain and port number, configure a URL based on a local private address and default port.
3. Instantiate a web view object.
4. To display the preview, send the URL with component name to the web view.

This sample also adds a bonus **Debug Info** button that reveals the launch argument configuration.

Where Do I Place the Configuration File?

To guarantee that the plug-in finds and uses your configuration file, follow these guidelines.

- **Requirement (all cases):** Name your configuration file `mobile-apps.json`. The plug-in doesn't accept other configuration file names.

Command-line usage

- Use the `preview` command's optional `-f` and `-d` parameters to specify a custom location:
  - `-f` specifies the config file's location. This path can be absolute or relative.
  - `-d` specifies the project's root directory.

The Mobile Extensions plug-in uses these two parameters as follows.

- If you specify `-f` but not `-d`, the plug-in uses the value of `-f` to determine the file's location.
- If you specify `-f` and `-d`, the plug-in can consider both values to determine the file's location.
  - If `-f` uses a relative path, the file location is calculated starting from the directory specified by `-d`. If `-d` isn't specified, the path is calculated against the current working directory.
  - If `-f` specifies an absolute path, the `-d` value is ignored in calculating the config file location.

Example 1

-f	`/../../mobile-apps.json`
-d	`/Users/jdoe/MyProject`
Result	`/Users/mobile-apps.json`

Example 2

-f	`/Users/jdoe/OtherProject/mobile-apps.json`
-d	Not specified
Result	`/Users/jdoe/OtherProject/mobile-apps.json`

Example 3

-f	`/Users/jdoe/OtherProject/mobile-apps.json`
-d	`/Users/jdoe/MyProject`
Result	`/Users/jdoe/OtherProject/mobile-apps.json`

Example 4

-f	`mobile-apps.json`
-d	Not specified
Result	`<current_working_directory>/mobile-app.json`

If the plug-in is unable to find the config file on the calculated path, it posts an error message showing the path that failed.

VS Code usage

In VS Code, you can't specify a custom path for the config file. VS Code always looks for `mobile-apps.json` in the default root folder of the Lightning Web Components project.

SEE ALSO:

Preview Components from the Command Line

Validate Lightning Web Components for Offline Use

Use the Komaci Static Analyzer (or static analyzer for short) during component development to validate your Lightning web components for offline use. Using the static analyzer helps you ensure that code dependencies and data your component depends on can be primed when a network connection is available, making the component and its data available offline when the network has limited or no connectivity.

IN THIS SECTION:

Install the Komaci Static Analyzer

The Komaci static analyser is an ESLint plugin that you install using a package manager, such as NPM or Yarn.

Troubleshoot Installation Problems

The Komaci Static Analyzer is implemented as a plugin for ESLint, a well-known JavaScript validation tool. ESLint plugins can be finicky in their installation and configuration, requiring that all pieces are perfectly aligned for success.

Validate Components During Development

To ensure your components can be used in offline environments, watch for and act on the recommendations of the static analyzer.

Static Analyzer Validation Rules

The static analyzer validates that various code constructs and references in Lightning web components, such as wire decorators and server calls, support offline priming. The majority of the rules focus on determining whether all code dependencies, such as imports and modules, can be resolved; or on correct usage of offline-compatible wire adapters.

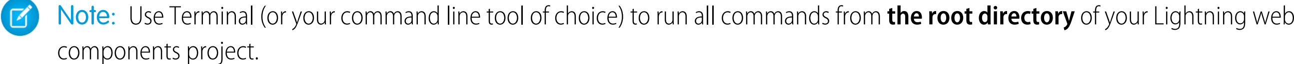
Install ESLint Rules for Mobile Lightning Web Components

We've created ESLint rules to help you develop code that works with mobile and offline Lightning web components. You can install them on your development machine and run them on your source code.

Install the Komaci Static Analyzer

The Komaci static analyser is an ESLint plugin that you install using a package manager, such as NPM or Yarn.

- Node.js >= 14.0.0
- A supported package manager
  - NPM >= 6.0.0
  - Yarn (Classic) >= 1.22.19



Note: Use Terminal (or your command line tool of choice) to run all commands from **the root directory** of your Lightning web components project.

1. Add the Komaci Static Analyzer plugin and its dependencies to the development dependencies of your project.

   NPM

   ```
   npm install --save-dev @salesforce/eslint-plugin-lwc-graph-analyzer
   ```

   YARN

   ```
   yarn add --dev @salesforce/eslint-plugin-lwc-graph-analyzer
   ```

2. Install all project modules and dependencies locally in the project.

NPM

```
npm install
```

YARN

```
yarn install
```

3. Configure your project to use the new plugin.

   Modify the `.eslintrc.json` file under your project's `force-app/main/default/lwc` directory by adding the bolded text:

```
{

  "extends": [
    "@salesforce/eslint-config-lwc/recommended",

    "plugin:@salesforce/lwc-graph-analyzer/recommended"
  ],
  "overrides": [
...
```

Troubleshoot Installation Problems

The Komaci Static Analyzer is implemented as a plugin for ESLint, a well-known JavaScript validation tool. ESLint plugins can be finicky in their installation and configuration, requiring that all pieces are perfectly aligned for success.

Unfortunately, there's often not much feedback—if any—if something went wrong in the process. The most likely clue that there's a problem is when known errors in your code produce no feedback.

If you're not sure if the static analyzer or ESLint has been successfully configured, open an LWC JavaScript file where you would expect linting feedback. Then, check the ESLint section of the Output panel in VS Code for feedback messages. To open the Output panel, click View Output. Then, to open the ESLint logs, select ESLint from the dropdown in the upper right corner of the Output panel.

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE          ESLint
[Info  - 11:02:10 AM] ESLint server is starting
[Info  - 11:02:11 AM] ESLint server running in node v14.16.0
[Info  - 11:02:12 AM] ESLint server is running.
[Info  - 11:02:13 AM] ESLint library loaded from: /Users/devuser/OfflineLintingDemo/node_modules/eslint/lib/
api.js
[Info  - 11:02:13 AM] Failed to load config "@salesforce/eslint-config-lwc/recommended" to extend from.
Referenced from: /Users/devuser/OfflineLintingDemo/force-app/main/default/lwc/.eslintrc.json
```

Common Issues

- **Install into your project** — For the validation package to function properly, you must install all dependent packages using the commands in the installation instructions from the root directory of your project. For example, the error in the preceding Output panel screenshot indicates a missing package. This is due to not installing a package required for the project.
- **Check your `.eslintrc.json` file** — If you misconfigure your `.eslintrc.json`, the ESLint Output panel might show an error related to that file. Ensure that the lines you add to your `.eslintrc.json` look exactly like the example in the installation instructions and fix any syntax errors.

- **ESLint must be authorized** — If you don't see any feedback in the ESLint output view, the ESLint server itself is likely not getting initialized for your project. Make sure that you have:
  - Installed theESLint extension for Visual Studio Code.
  - Installed project dependencies into your project with `npm install` (or `yarn install`) from your project's root directory. The command installs the ESLint package and other dependencies locally to your project.
  - Authorized ESLint to run in your project. If it's not, you might see an "error" squiggly line in the first line of your JavaScript file, indicating that ESLint isn't authorized. Hovering over that error gives you the option to authorize it for your project.

Validate Components During Development

To ensure your components can be used in offline environments, watch for and act on the recommendations of the static analyzer.

As you develop your Lightning web components, validation indicators, displayed as colored squiggly lines, appear under code to highlight that it violates a validation rule. Red lines indicate an error, and yellow lines indicate a warning.

```
@wire(getRecordAlternative, {
  recordId: "$recordId",
  layoutTypes: "Compact"
})
record;
```

To fix an issue, hover over the problematic code. A pop-up appears near the code, describing what the problem is and how to fix it.

```
The wire adapter cannot be primed because it refers to a missing
resource eslint(@salesforce/lwc-graph-analyzer/no-missing-resource-cannot-
prime-wire-adapter)
View Problem (⌥F8)   Quick Fix... (⌘.)
@wire(getRecordAlternative, {
  recordId: "$recordId",
  layoutTypes: "Compact"
})
record;
```

- You'll see a description of the issue, along with the linting package that found the issue.
- Click **View Problem** to highlight the code causing the issue, with a box underneath describing the issue and what linting package found the issue.
- Click **Quick Fix** to show a dropdown of options of how to fix the issue. Then, select an option from the menu to automatically perform that fix.

Static Analyzer Validation Rules

The static analyzer validates that various code constructs and references in Lightning web components, such as wire decorators and server calls, support offline priming. The majority of the rules focus on determining whether all code dependencies, such as imports and modules, can be resolved; or on correct usage of offline-compatible wire adapters.

Using a wire provides an efficient and mostly transparent mechanism to deliver required data for a given set of Lightning web components. Today, not all wire adapters are enabled for offline use, and even those that are must be used correctly to work while offline. Without a properly formatted wire, the number of network requests increases, components render slowly, and performance suffers.

For offline support, mobile apps at Salesforce use these wires to ensure the correct data for each record and its components are always ready to use. A wire that doesn't align with the validation rules defined in this package can result in features implemented with Lightning web components not working correctly when offline.

For more information on wires and how to use them, see Use the Wire Service to Get Data in the *Lightning Web Components Developer Guide*.

Install ESLint Rules for Mobile Lightning Web Components

We've created ESLint rules to help you develop code that works with mobile and offline Lightning web components. You can install them on your development machine and run them on your source code.

The ESLint rules flag violations for:

- Apex usage
- Offline GraphQL feature limitations
- Offline GraphQL hard limits

The ESLint rules are documented in the ESLint Plugin LWC Mobile GitHub repository.

The ESLint rules are a plugin that you install using a package manager, such as NPM or Yarn.

- Node.js >= 14.0.0
- A supported package manager
  - NPM >= 6.0.0
  - Yarn (Classic) >= 1.22.19

Note: Use Terminal (or your command line tool of choice) to run all commands from **the root directory** of your Lightning web components project.

1. Install the node project dependencies.

   NPM

   ```
   npm install --save-dev @salesforce/eslint-plugin-lwc-mobile
   ```

   YARN

   ```
   yarn add --dev @salesforce/eslint-plugin-lwc-mobile
   ```

2. Configure your project to use the new plugin.

   Modify the `.eslintrc.json` file under your project's `force-app/main/default/lwc` directory. Extending the `plugin:@salesforce/lwc-mobile/recommended` ESlint configuration enables static analysis of all `.js` files used in your Lightning web components.

   ```
   {
       "extends": ["eslint:recommended", "plugin:@salesforce/lwc-mobile/recommended"]
   }
   ```

IN THIS SECTION:

Use ESLint Rules in Visual Studio Code

The ESLint rules for warnings against Lightning web components are displayed in Visual Studio (VS) Code where your code violates them. The rules map to distortions that affect your code. The popup for a rule violation includes a link to documentation for the rule.

Use ESLint Rules in Visual Studio Code

The ESLint rules for warnings against Lightning web components are displayed in Visual Studio (VS) Code where your code violates them. The rules map to distortions that affect your code. The popup for a rule violation includes a link to documentation for the rule.

Note: Salesforce recommends using Visual Studio Code with the Salesforce Extensions for Visual Studio Code to develop offline Lightning web components.

The ESLint rules are documented in the ESLint Plugin LWC Mobile GitHub repository.

- Apex usage rules
- Offline GraphQL rules

Here you can see the popup for a lint rule violation.

```
161                     Id
162                   }
163                 }
164               }
165             }
166           }
167         }
168       `;
169     }
170
171     get testSemiJoin() {
172       return gql`
173         query AccountExample {
174           uiapi {
175             query {   Offline GraphQL: "Semi" join is not supported for mobile offline. eslint(@salesforce/lwc-mobile/offline-graphql-no-
176               Accoun  semi-anti-join-supported)
177                 wher
178                   Id  View Problem (⌥F8)    Quick Fix... (⌘.)
179                     id: {
180                       Opportunity: { StageName: { eq: "Closed Won" } }
181                       ApiName: "AccountId"
182                     }
183                   }
184                 }
185               ) {
186                 edges {
187                   node {
188                     Id
189                     Name {
190                       value
191                     }
192                   }
193                 }
194               }
```

This popup shows a lint warning, not an error.

For more information on how to use Apex and GraphQL while mobile and offline, see Use Apex While Mobile and Offline and Use GraphQL While Mobile and Offline in the *Mobile and Offline Developer Guide*.

Develop Offline-Ready LWCs with the LWC Offline Test Harness

The LWC Offline Test Harness (Test Harness, for short) is a lightweight testing, debugging, and inspection app. It enables developers to debug Lightning web components for use in their LWC Offline-based mobile apps. Use Test Harness to execute Quick Actions on selected SObjects from your Salesforce org, debug component JavaScript, and inspect drafts and draft queue behavior.

Test Harness provides error, warning, and info logs for your LWCs as it loads, runs, and interacts with Salesforce. View logging details for your data sync via the drafts queue, inspect Lightning logs from Debug Console, and attach Chrome and Safari debuggers to view the JavaScript console of the webview your LWCs run in.

Test Harness helps you confirm that your LWCs work as expected in LWC Offline-based environments and are ready for integration testing within an offline-enabled Salesforce mobile app.

Features

- Uses the latest version of LWC Offline, with all mobile capabilities, including those in developer preview.
- Quick and convenient app flow, centered around launching LWC quick actions with a selected SObject.
- Visible draft queue, for viewing the status of pending data modification operations.
- Debug Console embedded into the app, for both a broad view of ongoing tasks and granular inspection of log messages.
- Immediate, on-demand app reloads for quickly re-bootstrapping and re-running your latest LWC code changes.
- Attach browser debuggers to view more developer-specific errors and warnings from the LWC webview.

IN THIS SECTION:

Test Harness Overview

Learn the major features and where to find them in the Test Harness app.

Install the Test Harness App

Test Harness is distributed as an installable app package, not via the Apple App Store or Google Play Store. As a developer tool, it's intended to be installed into and used with a working development environment. The installation process, prerequisites, and compatibility details are consequently a bit more involved than with a normal mobile app.

Use the Test Harness App

Learn how to use the Test Harness to perform common testing and debugging actions.

Debug Lightning Web Components

The best way to develop and debug your Lightning web components is the same way you develop and debug *anything* built with HTML, CSS, and JavaScript: with the built-in debugging tools in your web browser.

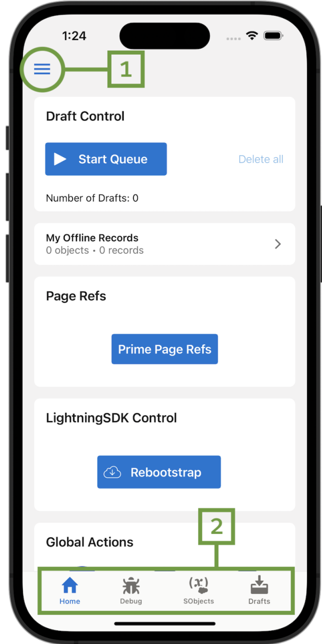
Test Harness Overview

Learn the major features and where to find them in the Test Harness app.

Test Harness is a general purpose development tool for mobile developers working on the Salesforce platform. It has a number of specific, purpose-built tools for inspecting the LWC Offline environment. It also makes it easy to run and inspect custom components under development. Some features look like they duplicate or overlap each other at a high level, but have important differences when you get into the specific behavior. Understanding which tool to use for a given purpose is essential to being successful with the tool.

There are two separate navigation controls for Test Harness.

- 1 — Sidebar menu ("hamburger" icon)
- 2 — Tabs bar



The sidebar menu provides access to a few duplicate or secondary features of Test Harness. Documentation for secondary features is forthcoming.

The tabs bar along the bottom of the app's user interface provides access to the primary features of Test Harness.

IN THIS SECTION:

Test Harness Home Tab

The Home tab of the Test Harness app is your home base for using the tool in your daily development activities. This screen is where you start from after you log into Salesforce with the app. It provides direct access to the most essential tools in Test Harness.

Test Harness Debug Tab

The Debug tab of the Test Harness app is a developer-centric tool for inspecting network logs and starting and stopping the offline draft queue.

Test Harness SObjects Tab

The SObjects tab of the Test Harness app is a developer-centric tool for examining Salesforce records, and the actions available on them.

Test Harness Drafts Tab

The Drafts tab of the Test Harness app is a developer-centric tool for controlling and examining the contents of the Offline Queue, including drafts you've created while offline.

Test Harness Home Tab

The Home tab of the Test Harness app is your home base for using the tool in your daily development activities. This screen is where you start from after you log into Salesforce with the app. It provides direct access to the most essential tools in Test Harness.

The Home tab is organized as a series of cards. Scroll to the card with the feature or tool you want to use. Some cards offer controls right on the card itself, while others lead you to secondary screens when you tap on them.

Test Harness comes with a predefined set of cards on the **Home** tab. Cards, such as **My Offline Records**, have a static user interface. Others, such as **Global Actions**, are controlled by metadata, and can be configured in Setup in your org.

My Offline Records

The **My Offline Records** card matches the same card in the Offline App in Salesforce Mobile App Plus.

Note: Your organization must purchase and license Salesforce Mobile App Plus in order to use the enhanced features. Contact your Salesforce sales rep for more information.

My Offline Records allows you to inspect the records that are primed by the Offline Briefcase assigned to your user. You can navigate into the list of objects that are included in the briefcase, and further into specific records.

The record view available in **My Offline Records** matches what your users see when using the Salesforce Mobile app in offline mode. That is, the view is driven by your org's metadata for page layouts, the way non-developers see them. For example, you see only the record-specific quick actions that are added to the page layout for the object, and each quick action has the icon you've defined for it. In contrast, when you view records in the developer-centric **SObjects** tab, **all** LWC-based quick actions defined for the object are displayed in a list, without icons.

Global Actions

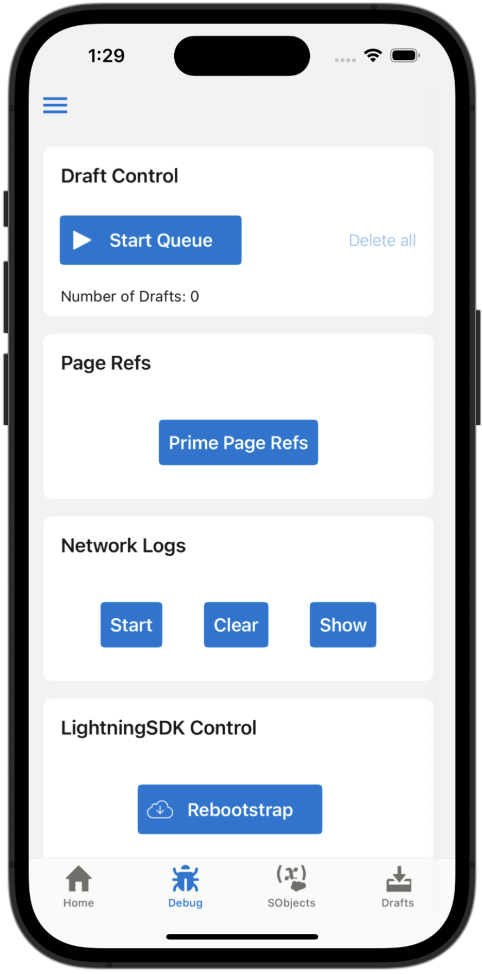
This card displays quick actions that you've added to a global publisher layout in your org. This makes it quick and easy to test global actions you're working on. See Add Actions to Global Publisher Layouts in the Salesforce Help for details of how to add your global actions to a publisher layout.

Recently Viewed Records

The **Home** tab displays the most recently accessed Account records in this card. The list is driven by activity, real record views in Salesforce—not viewing them in Test Harness. If you don't see the records you expect to see, view Salesforce in your desktop browser, and visit records of the appropriate type there. Then return to the Test Harness app, where you can pull-to-refresh to see an updated list of recently viewed records.

Test Harness Debug Tab

The Debug tab of the Test Harness app is a developer-centric tool for inspecting network logs and starting and stopping the offline draft queue.



Draft Control

Draft Control. As a result, you can perform actions when online, such as creating or editing a record, and give you a chance to inspect the results before they're uploaded to the Salesforce service. With the **Draft Control** card, you can only start and stop the queue and see how many drafts are waiting in the queue. For more insight into the contents of the Offline Queue, see the **Drafts** tab.

The Test Harness app starts with the Offline Queue in a paused state. Drafts that you create while the queue is paused wait in the Offline Queue until you tap **Start Queue**. When the queue is running, the button label changes to **Stop Queue**. If an error occurs while uploading a draft, the queue is paused automatically. Once the queue is empty, it returns to a paused state.

Page Refs

This is an advanced feature. It allows you to provide a list of resources to be primed by the LWC Offline engine. Each resource is specified as a `PageReference` using JSON.

Warning: This tool is under development. It should be used only with guidance from a Salesforce representative.

Network Logs

This card allows you to capture and inspect details regarding the network requests made by your Lightning web components. The tool captures requests after you tap **Start**, and stops capturing them when you tap **Stop**. To see a list of captured requests, tap **Show**. The

list shows the type and URL of the request, and the response code and duration. Tap a specific request to see further details, such as the headers and body of the request and response.

LightningSDK Control

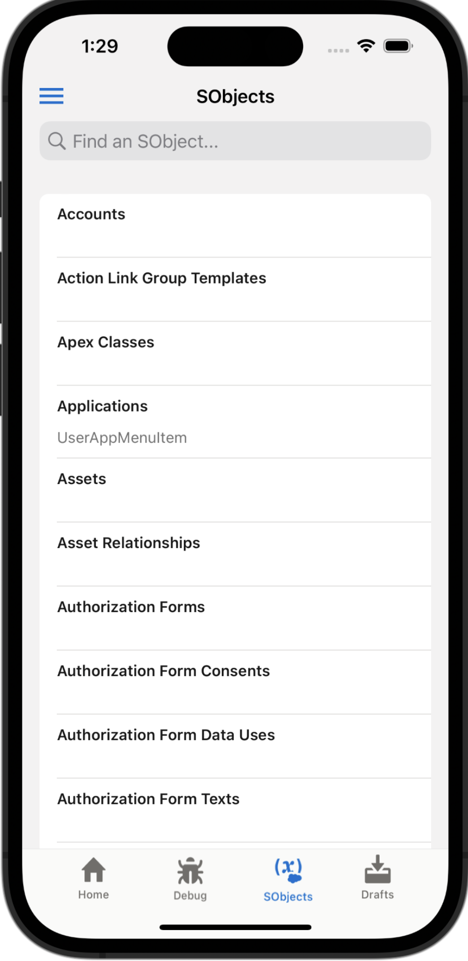
This card provides controls for affecting the LWC Offline engine. Currently, the only option is **Rebootstrap**. Use **Rebootstrap** to reload a component under development when you've made changes to the component's code. This allows you to quickly reload a custom component, without quitting and restarting the Test Harness app.

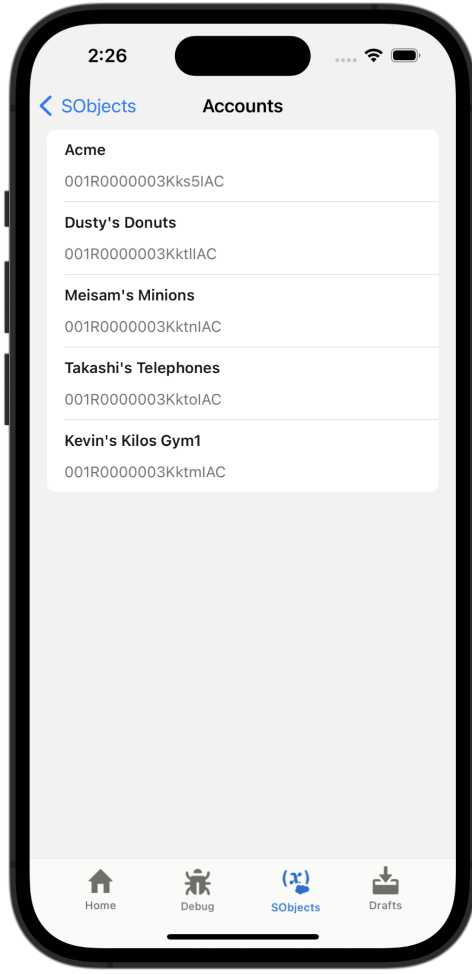
Test Harness SObjects Tab

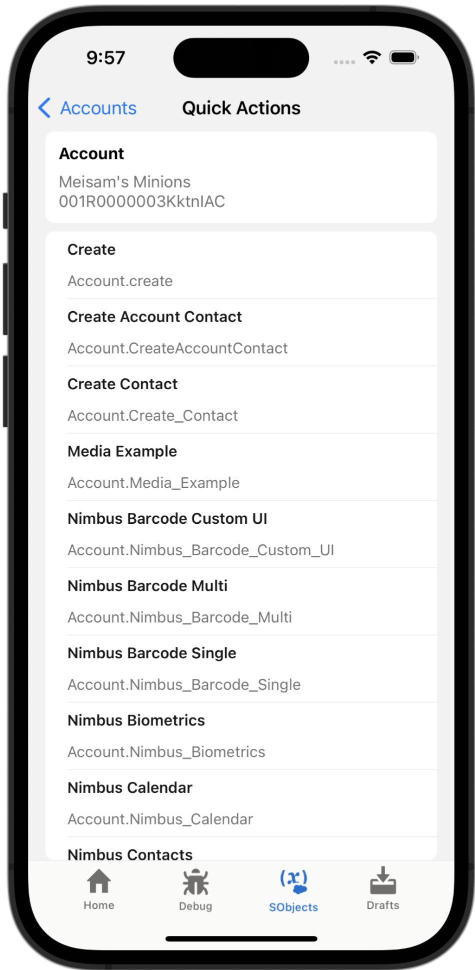
The SObjects tab of the Test Harness app is a developer-centric tool for examining Salesforce records, and the actions available on them.

The **SObjects** tab presents a plain list of the objects defined in your org. This list is driven by org metadata, and presents an unfiltered list of all of your objects. This list includes some objects that normally don't make sense in the context of the Test Harness app. For example, tapping the **Apex Classes** item displays an error.

Use the search field to filter the list to specific objects you're interested in working with. Tap an object name to see a list of records of that object type.




2:26
SObjects
Accounts
Acme
001R0000003Kks5IAC
Dusty's Donuts
001R0000003KktlIAC
Meisam's Minions
001R0000003KktnIAC
Takashi's Telephones
001R0000003KktoIAC
Kevin's Kilos Gym1
001R0000003KktmIAC
Home
Debug
SObjects
Drafts



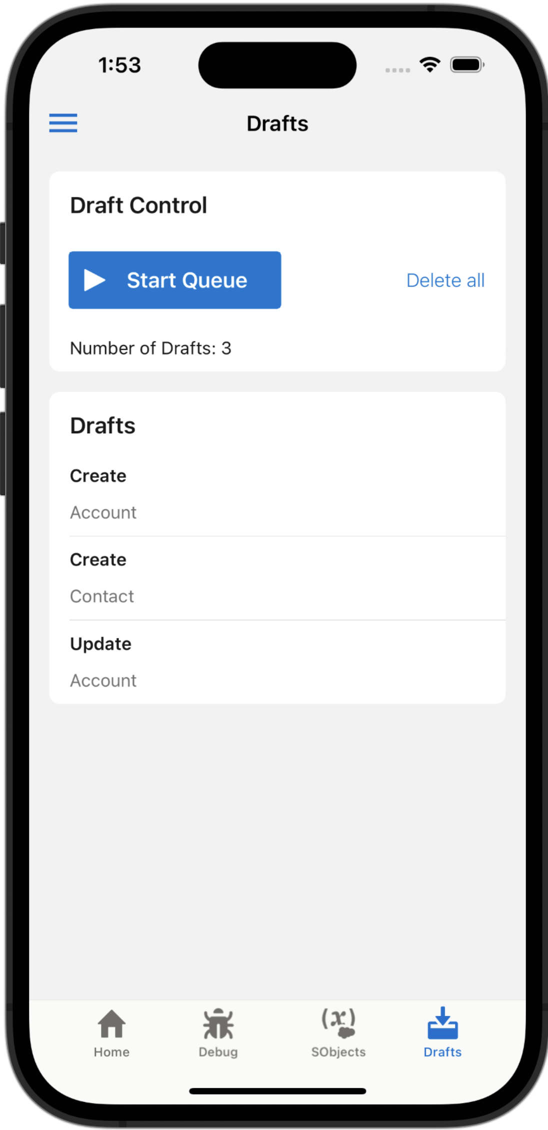
When you tap to view a record, minimal record data is displayed. The focus of this view is the list of actions available on the record. Tap an action to run it.

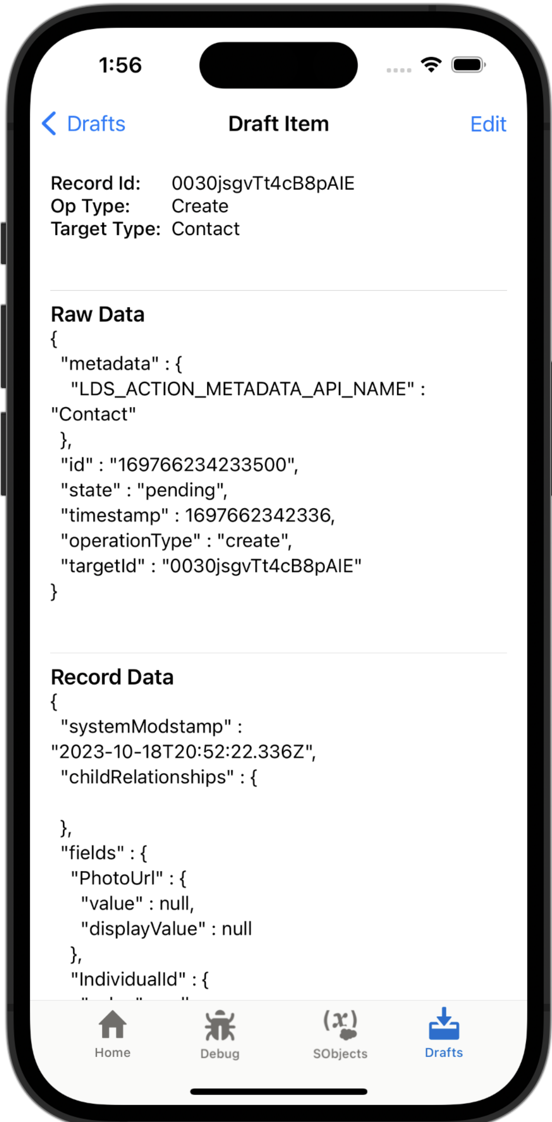
The list of actions are **all** of the LWC-based quick actions defined for the object, whether the actions have been added to the object's page layout or not. This makes the **SObjects** tab record view a great way to quickly view new quick actions under development, without the ceremony of adding them to a page layout, or the risk of appearing to users in your org before they're completed. In contrast, the record view available in **My Offline Records** presents a more end-user oriented view, and displays only the actions you've added to the object's page layout.

Test Harness Drafts Tab

The Drafts tab of the Test Harness app is a developer-centric tool for controlling and examining the contents of the Offline Queue, including drafts you've created while offline.

The **Drafts** tab displays the **Draft Control** card at the top, and then a list of the drafts currently in the Offline Queue. Tap a draft item to see the JSON data representation of the draft.


1:53
Drafts
Draft Control
Start Queue
Delete all
Number of Drafts: 3
Drafts
Create
Account
Create
Contact
Update
Account
Home
Debug
SObjects
Drafts



Draft Control. As a result, you can perform actions when online, such as creating or editing a record, and give you a chance to inspect the results before they're uploaded to the Salesforce service. With the **Draft Control** card, you can only start and stop the queue and see how many drafts are waiting in the queue.

The Test Harness app starts with the Offline Queue in a paused state. Drafts that you create while the queue is paused wait in the Offline Queue until you tap **Start Queue**. When the queue is running, the button label changes to **Stop Queue**. If an error occurs while uploading a draft, the queue is paused automatically. Once the queue is empty, it returns to a paused state.

Draft records in the Offline Queue are listed in the order they were created. They are local-only data, until you start the Offline Queue to upload the draft records to the Salesforce service.

Each item in the list can be more correctly described as representing a *draft operation*, or instructions for applying changes, either to create a new record or modify an existing one. The representation therefore includes details of what the operation is, and data for the record before and after the operation is applied.

Note: Documentation regarding the format and interpretation of draft records is forthcoming.

The **Edit** menu allows you to **Copy** the JSON representation of the draft into your clipboard. Copying can be useful for pasting into a code editor for detailed examination. You can also **Delete** the draft from the queue.

Install the Test Harness App

Test Harness is distributed as an installable app package, not via the Apple App Store or Google Play Store. As a developer tool, it's intended to be installed into and used with a working development environment. The installation process, prerequisites, and compatibility details are consequently a bit more involved than with a normal mobile app.

IN THIS SECTION:

Test Harness Prerequisites

To make full use of Test Harness, you need a complete working mobile development environment. If you haven't set up your mobile development tools yet, see the following resources for guidance.

Test Harness Compatibility

Test Harness is compatible with multiple versions of the Salesforce service. Some features of Test Harness can only be used with the latest release of Salesforce. Update older versions of Test Harness to the latest release.

Download and Install — Android

After your mobile development tools are up and running, getting started with Test Harness on Android devices is a breeze.

Download and Install — iOS

After your mobile development tools are up and running, getting started with Test Harness on iOS virtual devices is a breeze.

Test Harness Prerequisites

To make full use of Test Harness, you need a complete working mobile development environment. If you haven't set up your mobile development tools yet, see the following resources for guidance.

Salesforce DX Setup

- Set Up Your Development Environment for Lightning Web Components
- Set Up Your Development Environment for LWC Offline
- Preview Lightning Web Components on Mobile
- Set Up Xcode (required for iOS simulator)
- Set Up Android Studio (required for Android emulator)

Note: You don't need to set up both Xcode and Android Studio unless you want to use Test Harness in virtual devices on both platforms.

Third-Party Developer Tools

- Xcode Overview (Apple developer documentation)
- Meet Android Studio (Google developer documentation)

Test Harness Compatibility

Test Harness is compatible with multiple versions of the Salesforce service. Some features of Test Harness can only be used with the latest release of Salesforce. Update older versions of Test Harness to the latest release.

Test Harness Versions

The latest version of Test Harness can be downloaded by following the instructions provided.

- Download and Install — iOS
- Download and Install — Android

Salesforce Service Compatibility

For best results, always use the latest version of Test Harness with the latest release of the Salesforce Service. Don't mix versions of Test Harness and releases of Salesforce.

Mobile Device Platform Compatibility

On Android devices, the Test Harness app is distributed as an APK that can be installed on emulators or physical devices.

Minimum SDK version: API 28 (Pie)

On iOS devices, the Test Harness app is distributed as an app that can be installed on simulators. Test Harness can't be installed on physical devices.

Minimum iOS version: iOS 15.0

Download and Install — Android

After your mobile development tools are up and running, getting started with Test Harness on Android devices is a breeze.

1. Download the latest version of the Test Harness APK file.

   Important: Salesforce doesn't support releasing patch builds or bug fixes for older versions of the Test Harness APK file.

2. Open Android Studio and start an emulator compatible with Test Harness.

   For help with emulators in Android Studio, see Run an app on the Android Emulator (Google).

3. Using Finder (macOS) or File Explorer (Windows), navigate to your Downloads folder.
4. Find the Test Harness APK file you downloaded, and drag it onto a running Android emulator.

The Test Harness app is now installed on your emulated device.

Optional: You can download and install the Salesforce Offline Test Harness managed package to provide more control over OAuth authentication policies. See Install a Package in the Salesforce Help for more information on how to install a managed package.

Download and Install — iOS

After your mobile development tools are up and running, getting started with Test Harness on iOS virtual devices is a breeze.

1. Download the latest version of the Test Harness iOS app file.

   Important: Salesforce doesn't support releasing patch builds or bug fixes for older versions of the Test Harness iOS app.

2. Open Xcode and start a simulator compatible with Test Harness.

   For help with simulators in Xcode, see Getting Started in Simulator (Apple).

3. Using Finder, navigate to your Downloads folder.
4. Locate the Test Harness iOS app file you downloaded (its name is `LSDKTestHarness.app.zip`, or similar).

5. Unzip `LSDKTestHarness.app.zip` by double-clicking it. Then, find the newly extracted file in your Downloads folder, named `LSDKTestHarness.app`.
6. Drag the `LSDKTestHarness.app` onto a running iOS simulator.

The Test Harness app is now installed on your simulated device.

Optional: You can download and install the Salesforce Offline Test Harness managed package to provide more control over OAuth authentication policies. See Install a Package in the Salesforce Help for more information on how to install a managed package. Configure tabs available in the Field Service Mobile App.

Use the Test Harness App

Learn how to use the Test Harness to perform common testing and debugging actions.

IN THIS SECTION:

Create a Quick Action with an LWC
Quick actions are an easy way to add and launch your Lightning web components. It's simple to add them to the Test Harness app.

Display and Run an LWC from a Quick Action
During active development, use quick actions in the Test Harness app to launch and test your Lightning web components.

Create a Quick Action with an LWC

Quick actions are an easy way to add and launch your Lightning web components. It's simple to add them to the Test Harness app.

To create a Quick Action with an LWC, the LWC must have a target of type `lightning__RecordAction` defined in the component's `meta.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
  <LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>59.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
      <target>lightning__AppPage</target>
      <target>lightning__HomePage</target>
      <target>lightning__RecordAction</target>
    </targets>
  </LightningComponentBundle>
```

After a component is deployed to Salesforce, create a Quick Action that uses it. For details, see Quick Actions in the *Lightning Web Components Developer Guide*.

Tip: You don't need to add Quick Actions to a page layout to see it in Test Harness. The Test Harness app automatically shows all LWC-based Quick Actions for the displayed sObject type.

Display and Run an LWC from a Quick Action

During active development, use quick actions in the Test Harness app to launch and test your Lightning web components.

The Test Harness **SObjects** tab displays a list of all of your org's sObjects. Tap an sObject in the list to display up to 50 of the most recently viewed records for the selected sObject.

Tip: If the list is empty, log in to the org with the same user and view a few sample records you wish to test with. Those records are added to the Most Recently Viewed list in Salesforce. Return to Test Harness and access the sObject again by tapping on it. The recently viewed records appear in the list.

When you tap a record, you're presented with all of the LWC-based Quick Actions that are defined in **Buttons, Links, and Actions** in Setup for the selected sObject. Tap a Quick Action to launch the LWC.

SEE ALSO:

Test Harness SObjects Tab

Debug Lightning Web Components

The best way to develop and debug your Lightning web components is the same way you develop and debug *anything* built with HTML, CSS, and JavaScript: with the built-in debugging tools in your web browser.

- For debugging on Android, use Chrome DevTools (Google)
- For debugging on iOS, use Safari Web Inspector (Apple)

Desktop users use Salesforce in a standard web browser. For developers, this makes using the development tools built into those browsers the best way to experience the component code the same way their users do. It also makes it straightforward to use, examine, and debug code while it's under development. It all happens in the same tool, a standard web browser.

In contrast, mobile users don't use Salesforce in a browser, desktop or mobile. Instead, they use Salesforce, including the custom components you build, from within a Salesforce mobile app. Your LWCs run inside a *web view*, which is embedded in the mobile app. Web views don't have debugging tools built into them, so the process of debugging them is different.

The solution to this challenge is *remote debugging*. Remote debugging lets you use the exact same development and debugging tools you're used to—Chrome DevTools or Safari Web Inspector running on your development system—and connect them to the web view running inside a separate mobile app. Remote debugging works whether you're running your code in the app on a virtual device, or a physical device. While that all might sound complicated, it's actually simple once you understand the procedure.

IN THIS SECTION:

Enable Debugging for LWC Developers

Enable debugging settings for each LWC developer user who needs to use debugging tools while developing their LWCs.

Debug in Android

Follow these steps to attach Chrome DevTools in your browser to the webview of the Test Harness app.

Debug in iOS

Follow these steps to attach Safari Web Inspector in your browser to the webview of the Test Harness app.

Enable Debugging for LWC Developers

Enable debugging settings for each LWC developer user who needs to use debugging tools while developing their LWCs.

Debugging settings apply whether you're using Test Harness or other Salesforce mobile apps during debugging.

Enable Debug Mode

In normal use, Salesforce minifies and compresses JavaScript, HTML, CSS, and other assets for improved performance. These alterations can make it more difficult to debug your components. For example, tracing execution of your component's JavaScript code is much harder when the code has been minified.

The solution is to enable Debug Mode for users who are developing LWCs.

- In Salesforce, from Setup, enter `Debug Mode` in the Quick Find box, then select **Debug Mode Users**.
- In the user list, locate any users who need debug mode enabled.
- Enable the checkbox next to users for whom you want to enable debug mode.

More details, including a more complete description of the effects of Debug Mode, are available in "Enable Debug Mode in Salesforce" in the *Lightning Web Components Developer Guide*.

Debugging and Lightning Web Security

Lightning Locker and Lightning Web Security (LWS) enhance the security of your Salesforce org by enforcing certain rules about component behavior, partially isolating components from each other, and other measures. The effects on your component code can be significant, but are most likely to be problematic during debugging.

If, during debugging, your efforts are blocked by running into proxy objects, **temporarily** disable Lightning Web Security. This will allow you to access LWC objects directly, instead of via proxies.

For much more detail, see "Debug Components in an Org With LWS Enabled" in the *Lightning Web Components Developer Guide*.

SEE ALSO:

Lightning Web Components Developer Guide: Enable Debug Mode in Salesforce

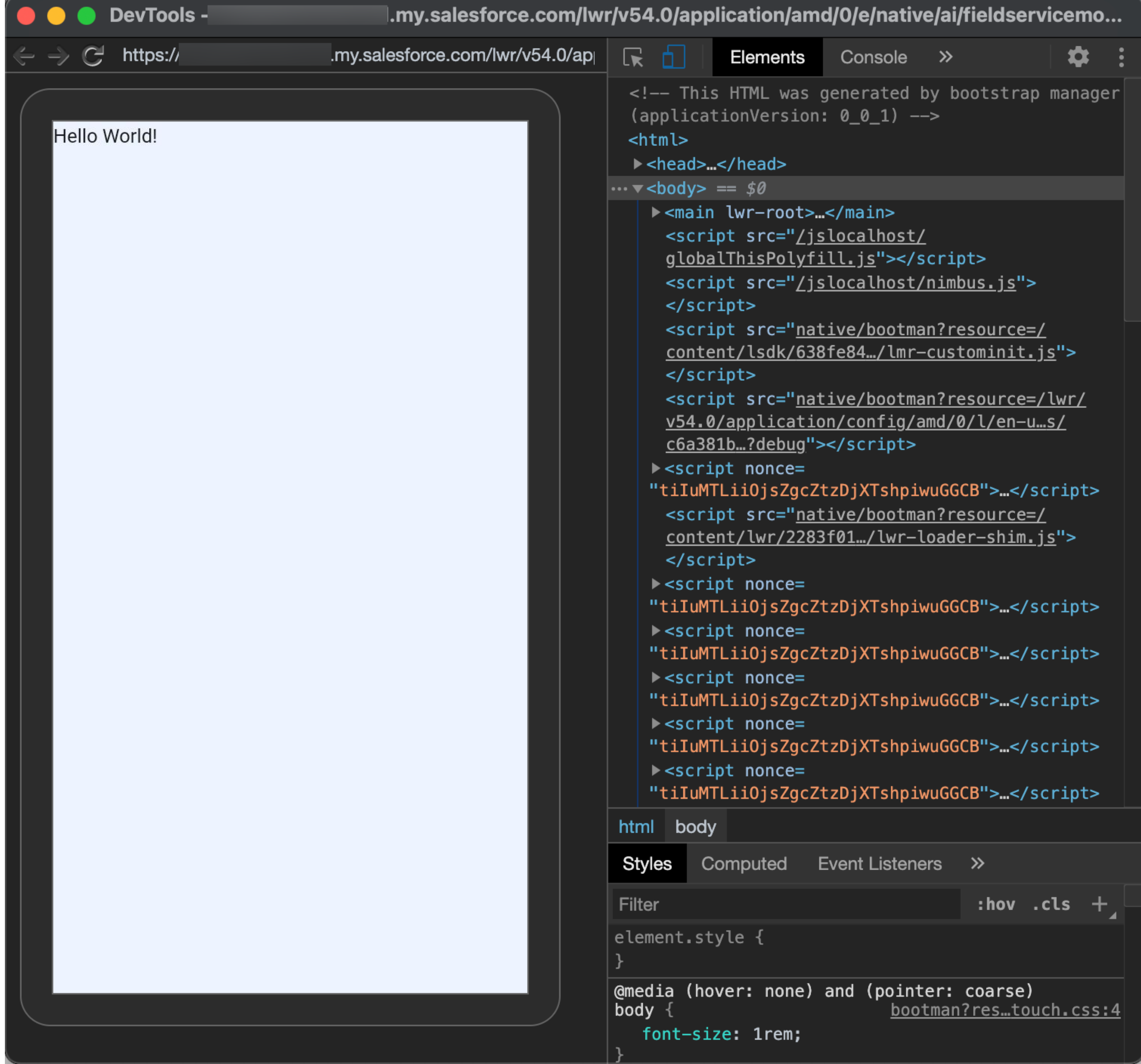
Lightning Web Components Developer Guide: Debug Components in an Org With LWS Enabled

Debug in Android

Follow these steps to attach Chrome DevTools in your browser to the webview of the Test Harness app.

1. On your emulated Android device, open the **Settings** app.
2. Enter `About emulated device` into the search bar, and tap the result titled **About emulated device**.
3. Scroll to the bottom of the page and tap **Build number** seven times. The message "You are now a developer!" appears after the seventh tap, meaning that developer mode is enabled for the emulator.

   You only need to do this step once for each emulated device you use for development.
4. Launch Chrome on your desktop. In the location bar, enter `chrome://inspect/#devices`.
5. Select **Inspect** for the WebView under the Remote Target emulator you're using.
   A window appears containing your emulator with Chrome DevTools connected to it. You can use Chrome DevTools to inspect the LWC element, set breakpoints, and see the console output. You can use standard web development techniques for working with HTML, CSS, and JavaScript to run, test, debug, and improve your LWC.



For additional information on the basics of debugging with Chrome Developer Tools, see Remote debugging WebViews (Google).

For a deeper dive on debugging JavaScript in an embedded WebView of an Android mobile app, see Debugging embedded JavaScript in an Android app using Chrome DevTools.

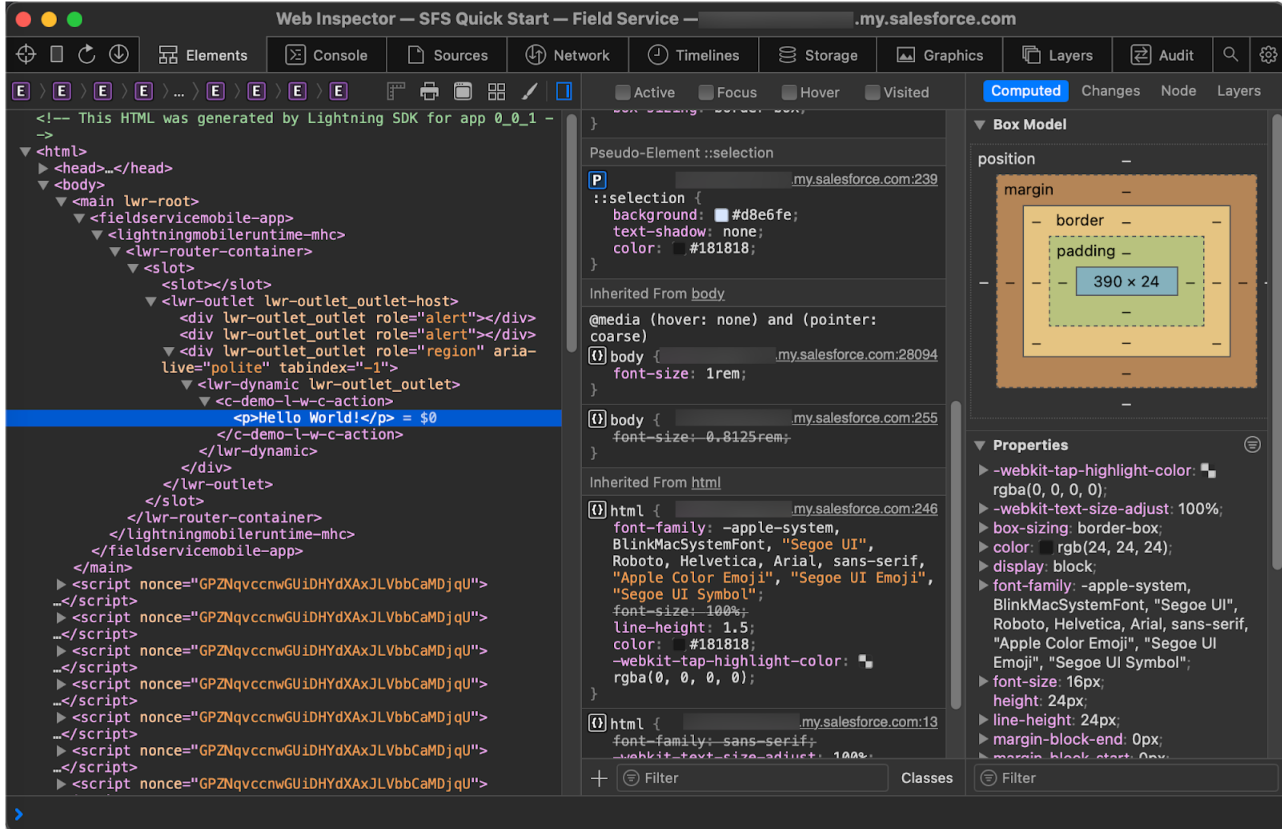
Debug in iOS

Follow these steps to attach Safari Web Inspector in your browser to the webview of the Test Harness app.

1. Launch Safari on your Desktop.
2. Select **Safari** > **Preferences**.
3. Select **Advanced**.
4. Enable **Show Develop menu in menu bar**. Then close the Preferences panel.

You only need to do this step once for your development system.

5. Select **Develop** > **Simulator - device - version**, where the Simulator is the one that you've opened with the Test Harness app for testing your new Lightning web component.
   A window appears, showing the Safari Web Inspector developer tools connected to your simulator. You can use these tools and standard web development techniques to refine and improve your content.



For additional information about the Web Inspector and how to use it, see Apple Web Development Tools (Apple).

Debug Your Components with Virtual Device Builds

To debug your components, connect your desktop browsers to the Salesforce mobile app running on your virtual devices. Then use Safari (iOS) or Chrome (Android) developer tools to view and interact with HTML markup, step through JavaScript code, and see console logging and error messages.

Note: You can debug the web-based code of your component, but you can't debug the platform native code of the mobile app.

Virtual device builds of the Salesforce mobile app let you preview your mobile Lightning web components on a wide range of simulated Android and iOS devices. To install a build into a simulated device, see Preview Components in the Salesforce Mobile App on page 192.

Debug on iOS Using Safari Developer Tools

To verify your component when running on iOS, use Safari Web Inspector.

To enable Safari's developer tools, including Web Inspector, follow these steps.

1. On your desktop development machine, open the Safari browser.
2. Select **Safari** > **Preferences…**.
3. Select **Advanced**.
4. Enable **Show Develop menu in menu bar**.

Safari developer tools are located in the **Develop** menu, and are active until you turn them off by reversing the preceding steps.

To debug a mobile component, connect your desktop instance of Safari to the Salesforce mobile app running on your virtual device.

1. On your desktop development machine, select **Safari** > **Develop** > **Simulator —** `device` **—** `version`, where `device` is the simulated hardware, and `version` is the operating system installed on it. For example, **Simulator — iPhone 11 Pro — iOS 13.3 (17C45)**.
2. Select **Automatically Show Web Inspector for JSContexts**.

When you interact with your component in the simulated device, a new Web Inspector window opens. Use Web Inspector to debug your component in much the same way you debug it when it's running directly in Safari on your desktop.

To learn more about using Safari Web Inspector, see webkit.org/web-inspector/enabling-web-inspector/.

Debug on Android Using Chrome DevTools

To debug your components when running on Android, use Chrome DevTools. First, enable Developer Mode on your emulated device.

1. On your emulated Android device, open **Settings** > **About Emulated Device**.
2. Scroll down to the build number. To enable Developer Mode, click the build number seven times. You see the message "You are now a developer!"
3. Return to the home screen, and open the Salesforce mobile app. Navigate to a page where your component is displayed.

To debug a mobile component, connect your desktop instance of Google Chrome to the Salesforce mobile app running on your virtual device.

1. In Chrome on your desktop development machine, enter this URL into the location bar: `chrome://inspect/#devices`.
2. You see a list of available remote debugging targets with names similar to "Android SDK built for x86_64 #EMULATOR-5554".
3. Find the item for your active emulated device. Under the Remote Target item, click **inspect**.

Chrome opens a remote debugging window. On the left is the current webview in the Android emulator, which is the page holding your mobile component. On the right is Chrome DevTools, which you can use to debug your component as if it were running in Chrome on your desktop.

To learn more about using Chrome DevTools, see developers.google.com/web/tools/chrome-devtools/remote-debugging.

SEE ALSO:

Salesforce Help: Salesforce Mobile App

Trailhead: Salesforce Mobile App Customization

Customize the Offline Experience for the Salesforce Mobile App

Mobile Offline is an advanced runtime environment for Lightning web components. Available only for mobile devices, it replaces the standard Lightning components runtime and augments it with features designed specifically for mobile and offline use.

Note: Your organization must purchase and license Salesforce Mobile App Plus to configure and use the Offline App. Contact your Salesforce sales rep for more information.

With the power of Lightning Web Components (LWC), you can create custom apps and experiences that tailor the Salesforce Mobile App to your specific business needs. LWC represents the best of the Salesforce platform, bringing modern web standards in performant, modular components that are easy to create. The goal of Mobile Offline with LWC is to help you build mobile experiences that work regardless of network conditions.

The Offline App is a pro-code solution that lets users configure LWCs for mobile offline experiences. The Offline App Developer Starter Kit, which is a publicly available GitHub repository, provides the resources to customize the offline experience. To make the development and set up process easier, we created a tool called the Offline App Onboarding Wizard. The Offline App Onboarding Wizard guides you with easy-to-follow prompts to configure the Offline Starter Kit experiences for the Offline App.

IN THIS SECTION:

Prerequisites & Setup Considerations

You need the correct tools installed to use the Offline App Onboarding Wizard. If you haven't set up your tools yet, see the following resources for guidance.

Download and Install

After you've gotten your development tools up and running, getting started with the Offline App Onboarding Wizard is a breeze.

Configure the Offline Experience

In this section, we provide supplemental information on configuring your offline experience.

Prerequisites & Setup Considerations

You need the correct tools installed to use the Offline App Onboarding Wizard. If you haven't set up your tools yet, see the following resources for guidance.

Salesforce DX Setup

- Salesforce CLI

Third-Party Developer Tools

- Visual Studio Code
- Salesforce Extension Pack
- Set up Git (GitHub)

To install and use additional tools specific for mobile and offline development, see Development Tools and Processes.

Download and Install

After you've gotten your development tools up and running, getting started with the Offline App Onboarding Wizard is a breeze.

1. Download the Salesforce Offline App Onboarding Wizard Visual Studio Code Extension. Alternatively, if you're in Visual Studio Code, go to the Extensions (Shift-Command-X on Mac) and search for the Salesforce Offline Starter Kit Wizard and click **Install**.
2. Open Visual Studio Code editor commands (Shift-Command-P or go to View | Command Palette) and select **Offline Starter Kit: Onboarding Wizard**.
3. Select **Create New Project** if this is your first time installing the Offline Starter Kit. Or select **Open Existing Project** to connect an existing project from your local folders.
4. Follow the Onboarding Wizard prompts to authorize an org and to set up your development environment.

Continue to use the Onboarding Wizard to set up and deploy your offline configurations.

Configure the Offline Experience

In this section, we provide supplemental information on configuring your offline experience.

For more detailed information, see about the Salesforce Offline App Onboarding Wizard Visual Studio Code Extension.

IN THIS SECTION:

1. Build a Briefcase

   Briefcase Builder is how you choose which records are available offline for your users. It's important to think about your users and which records they need access to while offline. The Offline App uses and depends on a Briefcase to use when priming records for offline use.

2. Select a Landing Page Template

   The Landing Page is the Mobile Offline home page for the Salesforce Mobile App. When a user logs into the Salesforce Mobile App offline experience, the landing page is the first thing they see. This page is intended to give access to the user's most important records and allow them to quickly and easily take relevant actions.

3. Generate and Configure LWC Quick Actions

   The Onboarding Wizard can generate missing LWC quick actions based on the sObjects that are configured in your landing page template selection.

4. Deploy Your Configurations

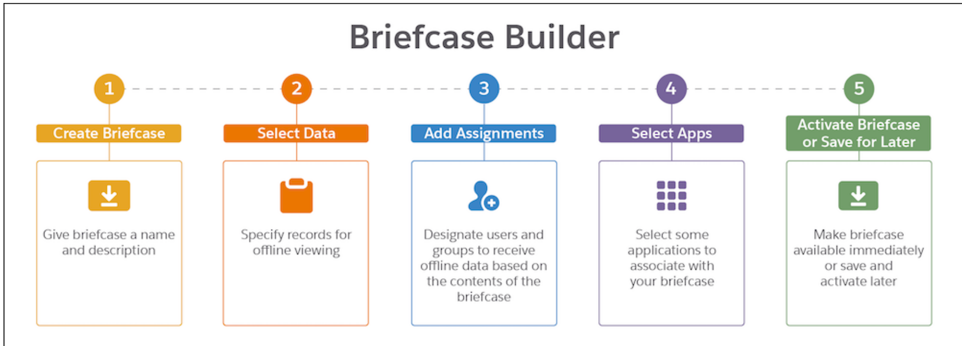
   Before you can run a quick action based on a Lightning web component, you need to deploy the relevant code artifacts to your org. Components and quick actions can be deployed using the Onboarding Wizard or manually with Visual Studio Code.

5. Add LWC Quick Actions to Mobile Layouts

   For a quick action to appear in the action bar of a record view, it must be assigned to the main page layout for the record's object type.

Build a Briefcase

Briefcase Builder is how you choose which records are available offline for your users. It's important to think about your users and which records they need access to while offline. The Offline App uses and depends on a Briefcase to use when priming records for offline use.



The Onboarding Wizard takes you to the Briefcase Builder page in Salesforce Setup so you can configure a briefcase for your org.

Tips For Briefcase Use With Mobile Offline

- Consider grouping users into specific roles and choosing records that are relevant for that user group.

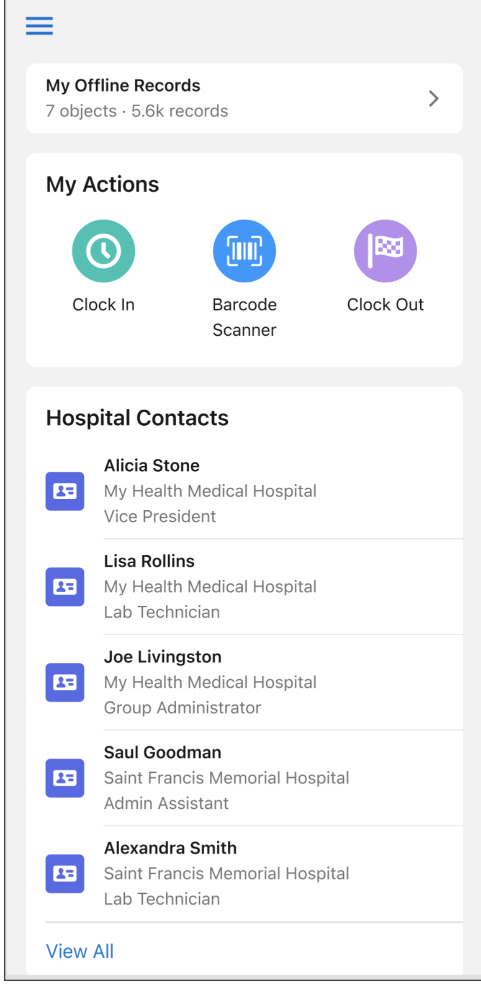
- Keep in mind that records that load on the Landing Page (Mobile Offline's home page) rely on the Briefcase to populate the record details.
- Ensure that all records that appear on a users Landing Page are also in their Briefcase.

For more information on building a briefcase, see Configure a Briefcase in the Salesforce Help.

Select a Landing Page Template

The Landing Page is the Mobile Offline home page for the Salesforce Mobile App. When a user logs into the Salesforce Mobile App offline experience, the landing page is the first thing they see. This page is intended to give access to the user's most important records and allow them to quickly and easily take relevant actions.

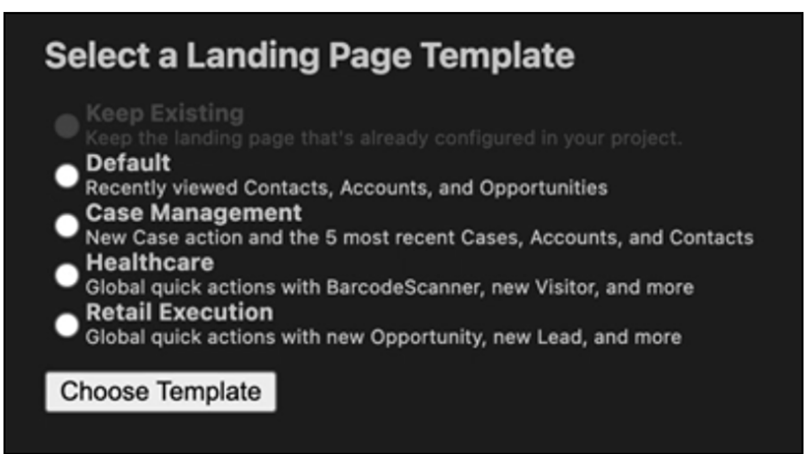
The landing page can be customized to meet user needs, using both out-of-the-box and custom objects, and Lightning global quick actions. It's configured as a static resource file in JSON format.



Note: Only one landing page layout is allowed per org.

Landing Page Templates

You can use the Onboarding Wizard to select a preconfigured landing page template that best fits your use case. The Onboarding Wizard copies the landing page JSON file (based on the chosen template) into the Offline App Starter Kit `staticresources` folder. Locate and update the `landing_page.json file` to build a customized offline app experience.



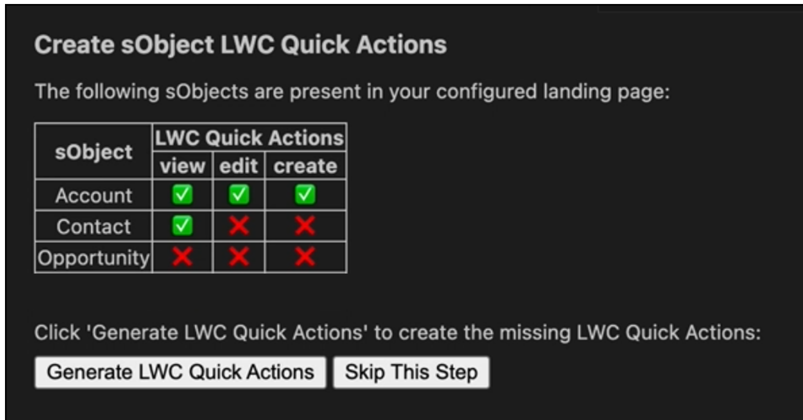
Each landing page template has specific objects and quick actions.

- Default
  - Accounts
  - Contacts
  - Opportunities
- Case Management
  - New Case action
  - 5 most recent Cases
  - 5 most recent Accounts
  - 5 most recent Contacts
- Healthcare
  - BarcodeScanner action
  - New Visit action
  - New Visitor action
  - Visit object
- Retail Execution
  - New Opportunity action
  - New Lead action
  - New Account action

For more information on updating the `landing_page.json` file, see Customize The Landing Page in the Salesforce Help.

Generate and Configure LWC Quick Actions

The Onboarding Wizard can generate missing LWC quick actions based on the sObjects that are configured in your landing page template selection.

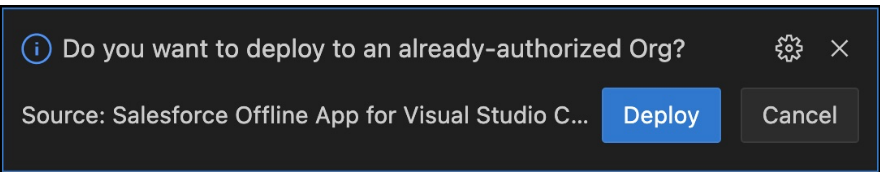


Review the missing LWC quick actions, and click **Generate LWC Quick Actions**.

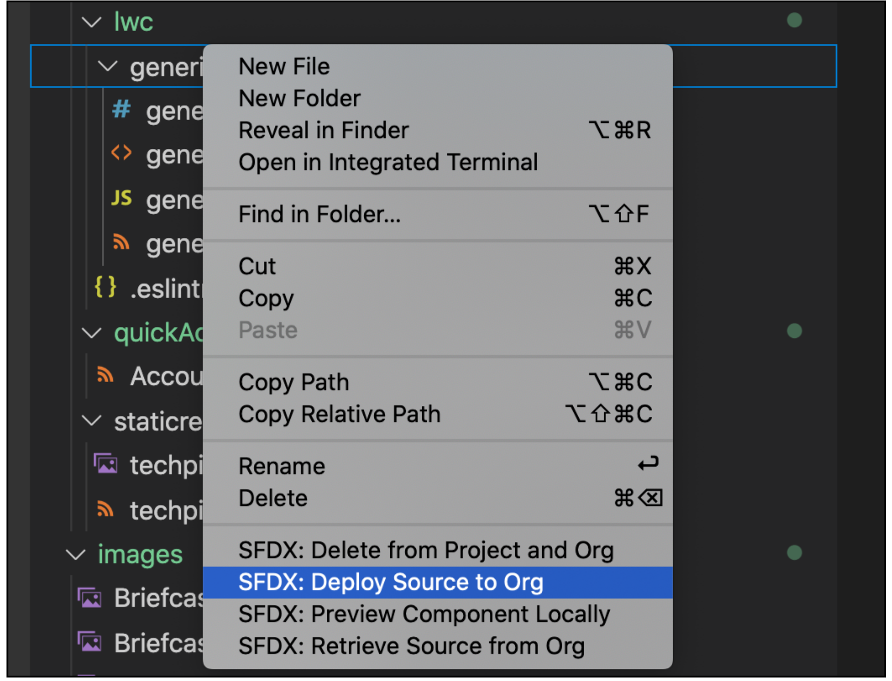
The Onboarding Wizard adds the generated quick actions to the Offline App Starter Kit `lwc` and `quickactions` folder.

Deploy Your Configurations

Before you can run a quick action based on a Lightning web component, you need to deploy the relevant code artifacts to your org. Components and quick actions can be deployed using the Onboarding Wizard or manually with Visual Studio Code.



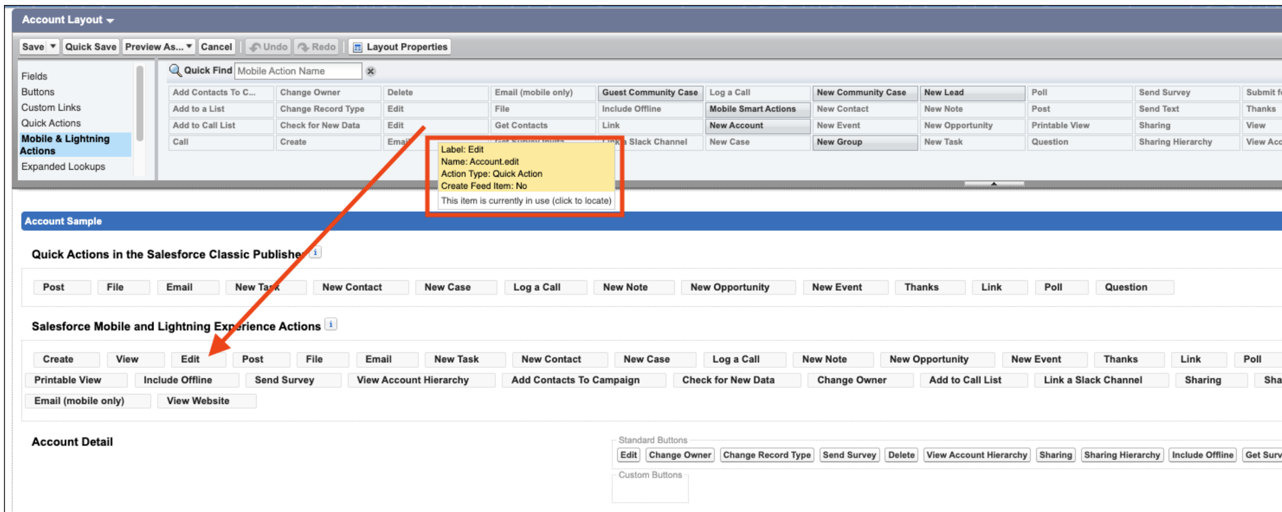
In Visual Studio Code, right-click on a component or quick action and select **SFDX Command: Deploy Source to Org**. This action pushes your Offline Starter Kit project configurations for the LWC Quick Actions upstream to the Salesforce org that's connected via your Visual Studio Code development environment.



Note: You might need to clear caches, and quit and restart the app before changes to LWCs are active.

Add LWC Quick Actions to Mobile Layouts

For a quick action to appear in the action bar of a record view, it must be assigned to the main page layout for the record's object type.



Here's an example of assigning the Edit quick action for the Account object type:

1. From Setup, open the **Object Manager**.
2. Enter `Account` in the Quick Find box, then select **Account**.
3. From the Account object management settings, go to **Page Layouts** and click **Account Layout**.

4. In the **Salesforce Mobile and Lightning Experience Actions** panel, if you see a link to override the predefined actions, the page layout is using the default actions. Click the link to enable customizing the actions.

CHAPTER 8 Quick Start Tutorials

In this chapter ...

- Develop a Lightning Web Component Quick Action
- Debug Lightning Web Components in the Field Service Mobile App

These hands-on tutorials get you started with creating custom LWCs for your mobile apps.

Develop a Lightning Web Component Quick Action

Welcome to developing quick actions using Lightning web components (LWCs) in the Salesforce Field Service (SFS) mobile app. With custom quick actions, you can tailor your Field Service mobile app users' experience to have easier access to viewing and updating information relevant to them.

This quick start guides you through the basic steps to create a custom record-specific quick action. It's a hands-on tutorial, intended to guide you through the steps, developing your "muscle memory" for a specific development lifecycle. While you *can* just read it, you'll get more out of it if you follow along on your own development system.

IN THIS SECTION:

Prerequisites

Ensure you're ready for this tutorial by verifying that you have the right software installed and configured, and your org has Lightning web components enabled for mobile users.

Field Service Org Setup

Create a permission set with the `Access Lightning Web Components in Field Service Mobile` permission, and assign this permission set to users who develop for or use Lightning web components in the mobile app.

iOS Simulator Setup

During development it's convenient to test your code in a virtual device. Use Xcode to create a device simulator, and install the virtual device build of your mobile app into it.

Android Emulator Setup

During development it's convenient to test your code in a virtual device. Use Android Studio to create a device emulator, and install the virtual device build of your mobile app into it.

Workspace Setup

Set up your development environment, create a project to develop your LWC in, and connect your project to Salesforce.

Create and Configure a Lightning Web Component

In this section, you'll create a basic component named `demoLWCAction` that displays a "Hello World!" message on the screen.

Prerequisites

Ensure you're ready for this tutorial by verifying that you have the right software installed and configured, and your org has Lightning web components enabled for mobile users.

- Salesforce CLI is installed and up-to-date.
- VS Code is installed and up-to-date.
- Salesforce Extension Pack for VS Code is installed and up-to-date.

Note: Ensure all prerequisites listed on the extension pack page are also satisfied.

Field Service Org Setup

Create a permission set with the `Access Lightning Web Components in Field Service Mobile` permission, and assign this permission set to users who develop for or use Lightning web components in the mobile app.

Note: This step is required for Field Service orgs. If you're using LWC Offline in the Salesforce mobile app, skip to the next step.

Lightning web components for LWC Offline-enabled mobile apps is an opt-in feature. To enable it for your org, you must create and assign a permission set for your mobile users that includes enabling the `Access Lightning Web Components in Field Service Mobile` permission. This permission set is required to access and run LWCs within the Field Service mobile app. (While not technically required to develop LWCs, you'll have a hard time developing components when you can't run them.)

You can use this one permission set for any number of Lightning web components and mobile users.

IN THIS SECTION:

Define a Permission Set for Your Org

Create a permission set that applies the permissions required to enable Lightning web components to users with the permission set.

Assign the Permission Set to a Mobile User

Assign the permission set that enables Lightning web components to users who must use or develop LWCs.

Define a Permission Set for Your Org

Create a permission set that applies the permissions required to enable Lightning web components to users with the permission set.

1. From Setup, enter *`Permission Sets`* in the Quick Find Box and select **Permission Sets**.
2. Click **New**.
   - For **Label**, enter *`Field Service - LWC Offline`*.
   - For **Description**, enter *`Assign to Field Service Mobile users to give them permission to run LWC actions`*.
   - For **License**, select **Field Service Mobile**.
3. Click **Save**.
4. In the Find Settings box, enter *Access Lightning Web Components in Field Service Mobile* and click it.
5. Click **Edit**.
6. Select the **Access Lightning Web Components in Field Service Mobile** checkbox.
7. Click **Save**.

Assign the Permission Set to a Mobile User

Assign the permission set that enables Lightning web components to users who must use or develop LWCs.

1. From Setup, enter *`Permission Sets`* in the Quick Find Box and select **Permission Sets**.
2. Select **Field Service - LWC Offline**, which is the new permission set.
3. Click **Manage Assignments**.
4. Click **Add Assignments**.
5. Select your mobile user's checkbox.

   For this quick start, the mobile user is **you**.
6. Click **Assign**.

iOS Simulator Setup

During development it's convenient to test your code in a virtual device. Use Xcode to create a device simulator, and install the virtual device build of your mobile app into it.

IN THIS SECTION:

[Configure Minimum Required iOS Simulator Settings]

Ensure your virtual device meets the minimum device and iOS version requirements to run your mobile app.

[Install the Field Service App for iOS]

To run the Field Service mobile app in an iOS simulator, download and install a virtual device build of the app. After it's installed, open the app and log into your development org.

Configure Minimum Required iOS Simulator Settings

Ensure your virtual device meets the minimum device and iOS version requirements to run your mobile app.

Review the minimum requirements for your mobile app.

- Requirements for the Salesforce Mobile App
- Field Service Mobile App Requirements

1. Download and install the latest version of Xcode. If you already have Xcode installed, there's no need to reinstall it.
2. Launch Xcode.
3. In the menu bar, select **Xcode** > **Open Developer Tool** > **Simulator**.
   A new program called Simulator opens, displaying a mobile screen.
4. Go to **File** > **Open Simulator** to choose your preferred device.

   If you want to create a simulator, go to **File** > **New Simulator** and follow the prompts.

After the device simulator launches you can close Xcode, but keep the Simulator app's window open to install the Field Service app in the next section.

Install the Field Service App for iOS

To run the Field Service mobile app in an iOS simulator, download and install a virtual device build of the app. After it's installed, open the app and log into your development org.

1. Download the latest iOS virtual device build of the Salesforce Field Service mobile app zip file.
2. Double-click the downloaded zip file to extract the app file.
3. Drag the downloaded `.app` file into the Simulator window.
4. In Simulator, click (a simulated tap in this case) the newly installed Field Service app to open it.

   If you don't see the app on the first page, it's probably installed on a different app page. You can swipe the page with your mouse to see additional pages to find the app.
5. Click (simulator tap, you get the idea) the Field Service app. Click **Get Started**.
6. Click the screen to run through the tutorial or click **Skip**.
7. Click **Log In**.
8. Click **I Agree** to agree to the Order Form Supplement agreement.

9. Click ⚙ and select a connection.

   Warning: If you're logging in as a community user for the first time, click + to add a new connection.

   - For **Host**, enter your org's URL in the following format: `https://[yourURL].my.salesforce.com`.
   - For **Label**, enter a nickname for your connection.

10. Click **Done**.
11. Enter your username and password for your org.
12. Click **Log In**.
13. Click **Allow** to allow the app to access your Salesforce information.
14. Click through the various permissions screens and allow them the appropriate access.
    When finished, you arrive at the app's home screen.

Android Emulator Setup

During development it's convenient to test your code in a virtual device. Use Android Studio to create a device emulator, and install the virtual device build of your mobile app into it.

IN THIS SECTION:

Configure Minimum Required Android Emulator Settings
Ensure your virtual device meets the minimum device and Android API version requirements to run your mobile app.

Install the Field Service App for Android
To run the Field Service mobile app in an Android emulator, download and install a virtual device build of the app. After it's installed, open the app and log into your development org.

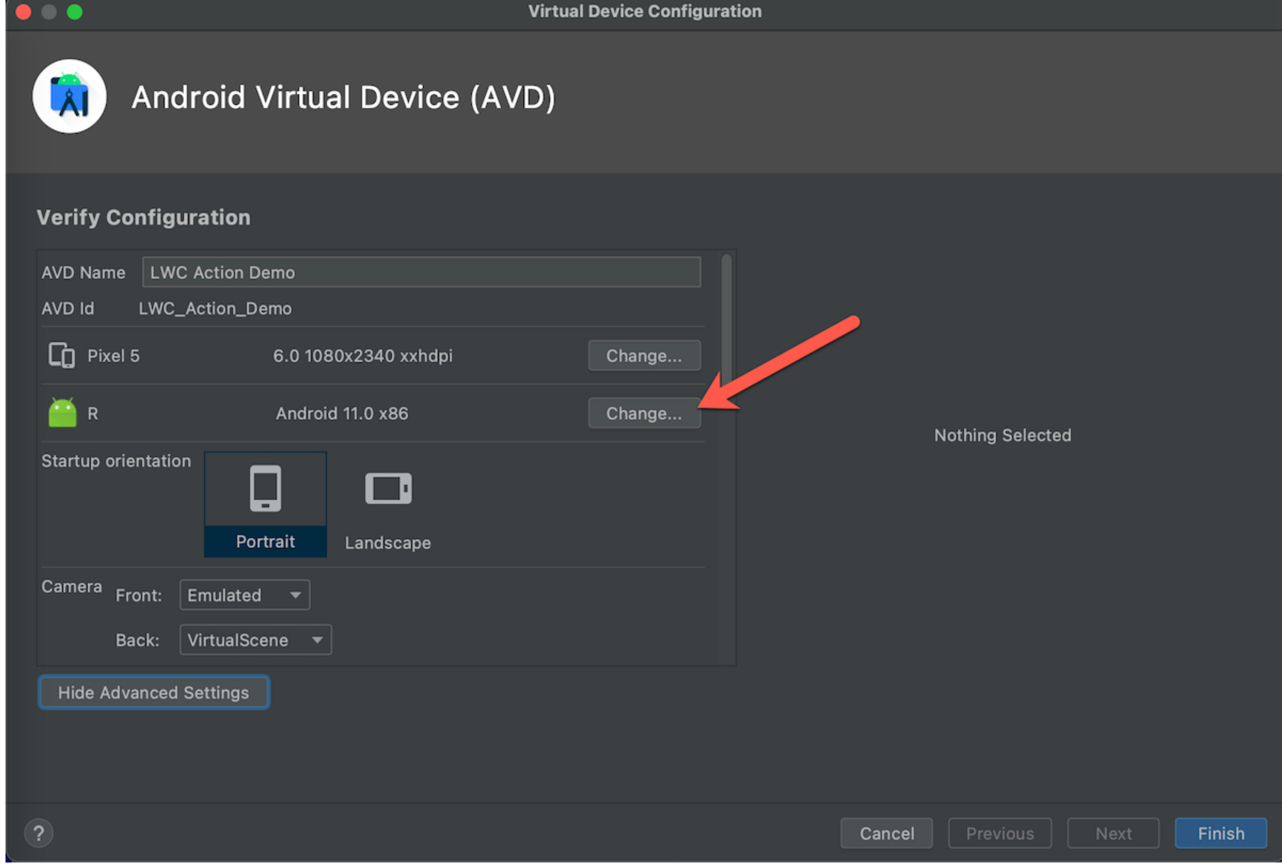
Configure Minimum Required Android Emulator Settings

Ensure your virtual device meets the minimum device and Android API version requirements to run your mobile app.

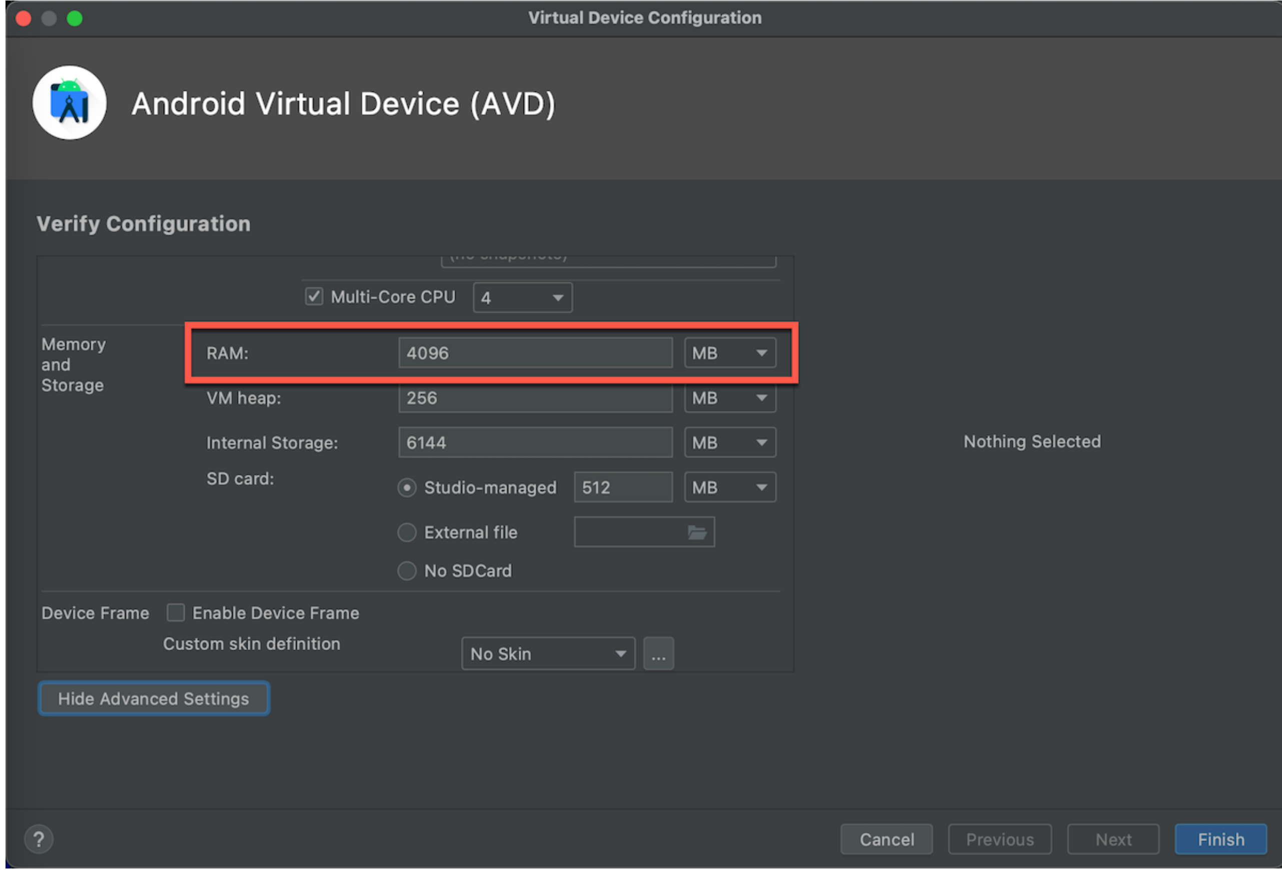
Review the minimum requirements for your mobile app.

- Requirements for the Salesforce Mobile App
- Field Service Mobile App Requirements

1. Download and install the latest version of Android Studio.
   If you already have Android Studio installed, there's no need to reinstall it.
2. Launch Android Studio.
3. Click **More Actions** or ⋮ in the top left, depending on your version of Android Studio, and then select **Virtual Device Manager** from the dropdown.
4. Click the ✎ in the Actions column of the device you'd like to edit.
   Or, if you want to create an emulator, click **Create Device** and follow the prompts.
5. Click **Change** in the line that displays the version number.



6. In the pop-up window, select API 30 version or later.
7. Click **OK**.
8. Scroll down in the window to the Memory and Storage section.



9. Click in the **RAM** field and enter `4096`.

   If you can't change the RAM value, use a newer device for the emulator.

10. Click **Finish**.

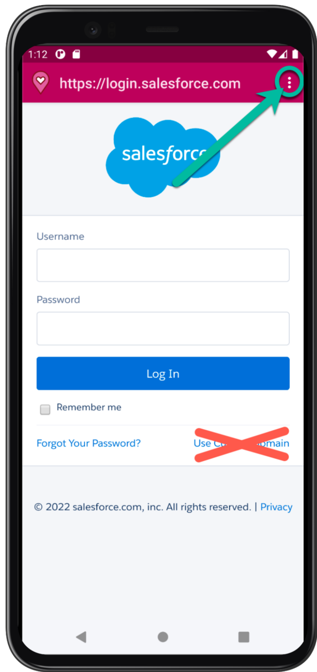
11. Click in the Actions column of the device to launch the Android Emulator.

After the device emulator launches you can close Android Studio, but keep the Android Emulator app window open to install the Field Service app in the next section.

Install the Field Service App for Android

To run the Field Service mobile app in an Android emulator, download and install a virtual device build of the app. After it's installed, open the app and log into your development org.

1. Download the latest Android virtual device build of the Salesforce Field Service mobile app APK file.
2. Drag the downloaded `.apk` file into the Android Emulator window.
3. Click a blank space on the Android Emulator's screen and drag up to view the installed apps.
4. Click the newly installed Field Service app.
5. Click **I Agree** to accept the Order Form Supplement agreement.
6. Click the vertical dots button on the top right and select **Change Server**.



7. Select a connection.

   Warning: If you're logging in as a community user for the first time, click **Add New Connection**, fill in the form as indicated in the following bullet list, and then click **Apply** to save the changes.

   - For **Name**, enter a nickname for your connection.
   - For **URL**, enter your org's URL. It must be in the following format: `https://[yourURL].my.salesforce.com`

8. Click the arrow in the top left to go back to the login screen.
9. Enter your username and password for your org.
10. Click **Log In**.
11. Click **Allow** to allow the app to access your Salesforce information.
12. Click through the various permissions screens and allow them the appropriate access.
    When finished, you arrive at the app's home screen.

Workspace Setup

Set up your development environment, create a project to develop your LWC in, and connect your project to Salesforce.

1. Launch VS Code.
2. Select **File** > **Open** and open a project to use with your org.

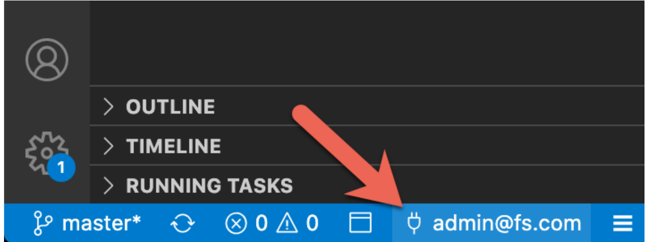
   Note: If you don't have a project created, you can open the VS Code Command Palette by clicking **View** > **Command Palette**, enter *`SFDX`*, and select **SFDX: Create a Project**. Then follow the prompts.

3. In the menu bar, select **View** > **Terminal** to open VS Code's integrated terminal if it's not already visible.

4. Run the following command to install the Salesforce Extension Plugins for mobile.

```
sf plugins install @salesforce/lwc-dev-mobile
```

5. In the menu bar, select **View** > **Command Palette** to open the VS Code Command Palette.
6. Enter `SFDX` and select **SFDX: Authorize an Org**.
7. Select the org you want to use and press **Enter**.
   A web page opens for the org login.
8. Verify your org authorization by checking the bottom left in VS Code.
   You should see your username in the VS Code status bar.



If you have difficulty authorizing SFDX for access to your org, see SFDX Authorization.

Create and Configure a Lightning Web Component

In this section, you'll create a basic component named `demoLWCAction` that displays a "Hello World!" message on the screen.

If this is your first Lightning web component ever, you might be tempted to rush through this. Take your time, and make sure you understand what each of these steps accomplishes.

IN THIS SECTION:

Create and Deploy a Lightning Web Component to Salesforce

Create a simple Lightning web component, configure its metadata, and then deploy the component to your Salesforce org—all from within VS Code.

Verify the Component Was Deployed to Your Org

The simplest way to verify that a Lightning web component is available in your org is to view the list of components in Setup in Salesforce.

Create a Quick Action in Salesforce

To access your Lightning web component, you must assign your component to a new Quick Action, and then assign the Quick Action to a page layout in Salesforce. This configuration makes it visible in the Actions launcher in the mobile app.

Add the Quick Action to a Page Layout in Salesforce

Make a Lightning web component quick action available in the mobile app by adding it to the mobile actions section of a page layout.

Clear Cached Metadata

To see changes to a Lightning web component as you develop, cached metadata must be cleared in the mobile app. Clear cached metadata every time new code is deployed to the org to see your changes.

Run the Quick Action in the Mobile App

Let's finally see that Lightning web component working in the Field Service mobile app.

Create and Deploy a Lightning Web Component to Salesforce

Create a simple Lightning web component, configure its metadata, and then deploy the component to your Salesforce org—all from within VS Code.

1. In the VS Code explorer, right-click the `force-app/main/default/lwc` folder and select **SFDX: Create Lightning Web Component**.
2. Enter *`demoLWCAction`* as the name for the component and press Enter, and then press Enter again to save it to the default location.
3. In the newly created `demoLWCAction.html`, replace the default code with the following, and then save the file:

```
<template>
    <p>Hello World!</p>
</template>
```

4. In the newly created `demoLWCAction.js-meta.xml`, replace the default code with the following and save the file:

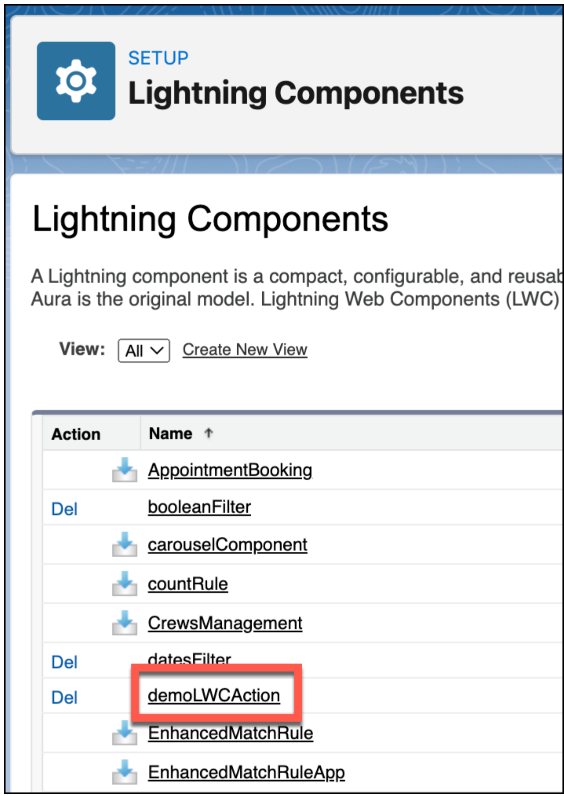
```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
       <target>lightning__RecordAction</target>
    </targets>
    <targetConfigs>
        <targetConfig targets="lightning__RecordAction">
            <actionType>ScreenAction</actionType>
        </targetConfig>
    </targetConfigs>
</LightningComponentBundle>
```

5. In the VS Code explorer, right-click the `force-app/main/default/lwc/demoLWCAction` folder and select **SFDX: Deploy Source to Org**.

Verify the Component Was Deployed to Your Org

The simplest way to verify that a Lightning web component is available in your org is to view the list of components in Setup in Salesforce.

1. From Setup, enter *`Lightning Components`* in the Quick Find Box and select **Lightning Components**.
2. Scroll through the list of components to see `demoLWCAction`.

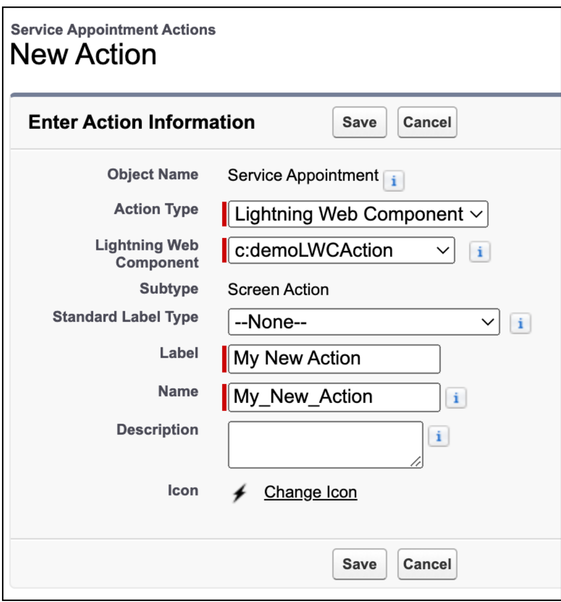


Create a Quick Action in Salesforce

To access your Lightning web component, you must assign your component to a new Quick Action, and then assign the Quick Action to a page layout in Salesforce. This configuration makes it visible in the Actions launcher in the mobile app.

For this example, the action is added to the Service Appointment object.

1. From Setup, click **Object Manager**.
2. In the Quick Find box, enter `Service Appointment`, and click **Service Appointment**.
3. Click **Buttons, Links, Actions**.
4. Click **New Action**.
   - For **Action Type**, select **Lightning Web Component**.
   - For **Lightning Web Components**, select **c:demoLWCAction**.
   - For **Label**, enter `My New Action`. This label is how the action is displayed in the Actions launcher in the Field Service mobile app.



5. Click **Save**.
6. Click **Service Appointment** next to **Object Name** to return to the Service Appointment object page.

Add the Quick Action to a Page Layout in Salesforce

Make a Lightning web component quick action available in the mobile app by adding it to the mobile actions section of a page layout.

1. From the Service Appointment object page, select **Page Layouts**.
2. Click the layout assigned to your mobile user.
3. Select **Mobile & Lightning Actions**.
4. Drag **My New Action** to the **Salesforce Mobile and Lightning Experience Actions** section.
5. Click **Save**.

Note: Mobile quick actions are only added to the mobile layout. You can't verify that mobile quick actions were successfully added to the correct page layout using the desktop experience.

Clear Cached Metadata

To see changes to a Lightning web component as you develop, cached metadata must be cleared in the mobile app. Clear cached metadata every time new code is deployed to the org to see your changes.

1. Launch the iOS Simulator or Android Emulator.
2. Open the Field Service mobile app.
3. Select the **Profile** tab in the navigation bar.
4. Click , then **Advanced Settings**, and then **Clear Cached Metadata**.
5. On iOS devices, click **OK** in the confirmation dialog and then click **OK** again.
6. On iOS, swipe away the app in the app switcher to close it, then reopen the app.
7. To close and reopen on Android, in Field Service app settings click **Force Quit**, then reopen the app.

Run the Quick Action in the Mobile App

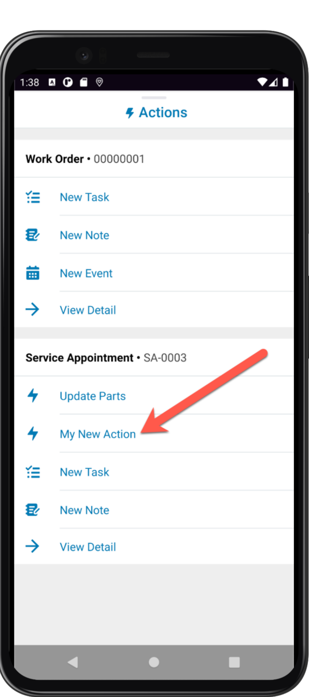
Let's finally see that Lightning web component working in the Field Service mobile app.

1. Click the **Schedule** tab.
2. Click any Service Appointment.

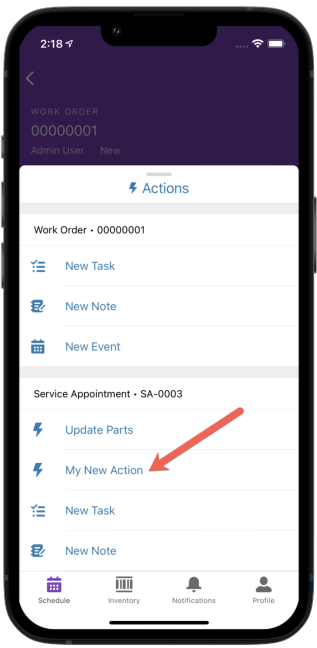
   If none are listed, create one and assign it to your mobile user as an example.

3. Click the **Actions** drawer and drag it up.
   Your action is listed as an option.

Android

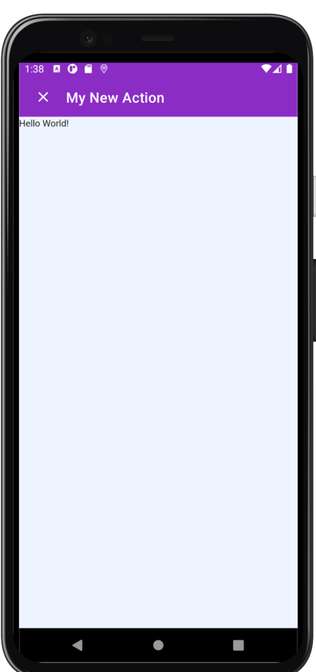


iOS

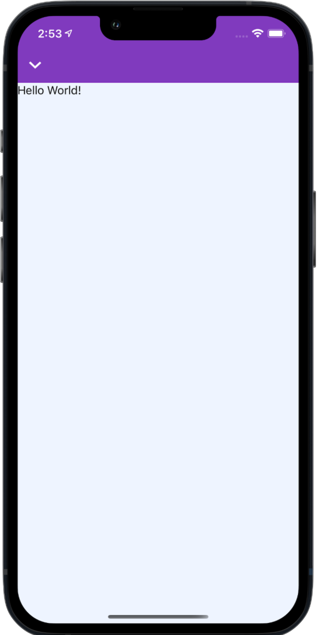


4. Click **My New Action**.
   This opens the screen to your new custom Lightning web component.

Android



iOS



Debug Lightning Web Components in the Field Service Mobile App

The best way to develop and debug your Lightning web components is the same way you develop and debug *anything* built with HTML, CSS, and JavaScript: with the debugging tools built into your web browser.

- For debugging on Android, use Chrome DevTools
- For debugging on iOS, use Safari Web Inspector

The rest of this tutorial guides you through connecting your desktop browser's developer tools to the WebView within the Field Service mobile app. From there, debugging an LWC running on a mobile device is like any other web app debugging session.

IN THIS SECTION:

Install Local Development Server Plugin

Whether you're debugging your component for iOS or Android, you must first install the LWC Development Server for mobile.

Debug in iOS

Connect Safari Web Inspector on your desktop to the WebView in the Field Service mobile app where your LWC is running.

Debug in Android

Connect Chrome DevTools on your desktop to the WebView in the Field Service mobile app where your LWC is running.

Install Local Development Server Plugin

Whether you're debugging your component for iOS or Android, you must first install the LWC Development Server for mobile.

1. In a terminal window in VS Code or Terminal, run the following command to ensure you're using the latest version of Salesforce CLI.

   ```
   sf update
   ```

Note: If you encounter an error when updating Salesforce CLI, see Update Salesforce CLI for troubleshooting instructions.

2. In the same terminal window, run the following command to install the LWC Development Server for mobile.

```
sf plugins install @salesforce/lwc-dev-server
```

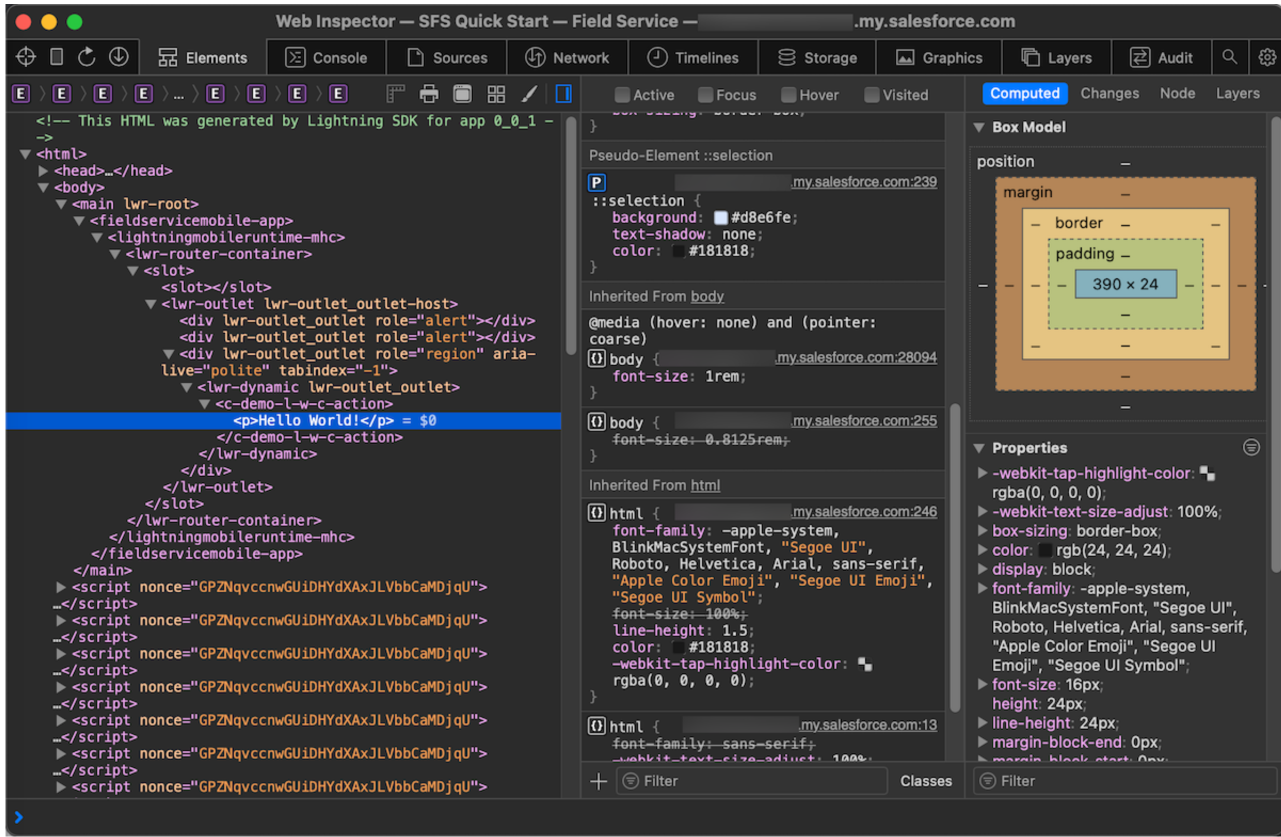
Debug in iOS

Connect Safari Web Inspector on your desktop to the WebView in the Field Service mobile app where your LWC is running.

Note: Debugging in iOS currently only works with Big Sur or later, and requires using the Safari Technology Preview browser.

1. Launch Safari on your desktop.
2. Select **Safari** > **Preferences**.
3. Select **Advanced**.
4. Enable **Show Develop menu in menu bar**. Close the Preferences panel.
5. Select **Develop** > **Simulator -** *`device`* **-** *`version`*, where the Simulator is the one that you've opened with the Field Service app for testing your new Lightning web component.

A window appears that shows the Safari Web Inspector developer tools connected to your simulator. You can use these tools and standard web development techniques to refine and improve your component.



See Apple Web Development Tools for additional information about the Web Inspector and how to use it.

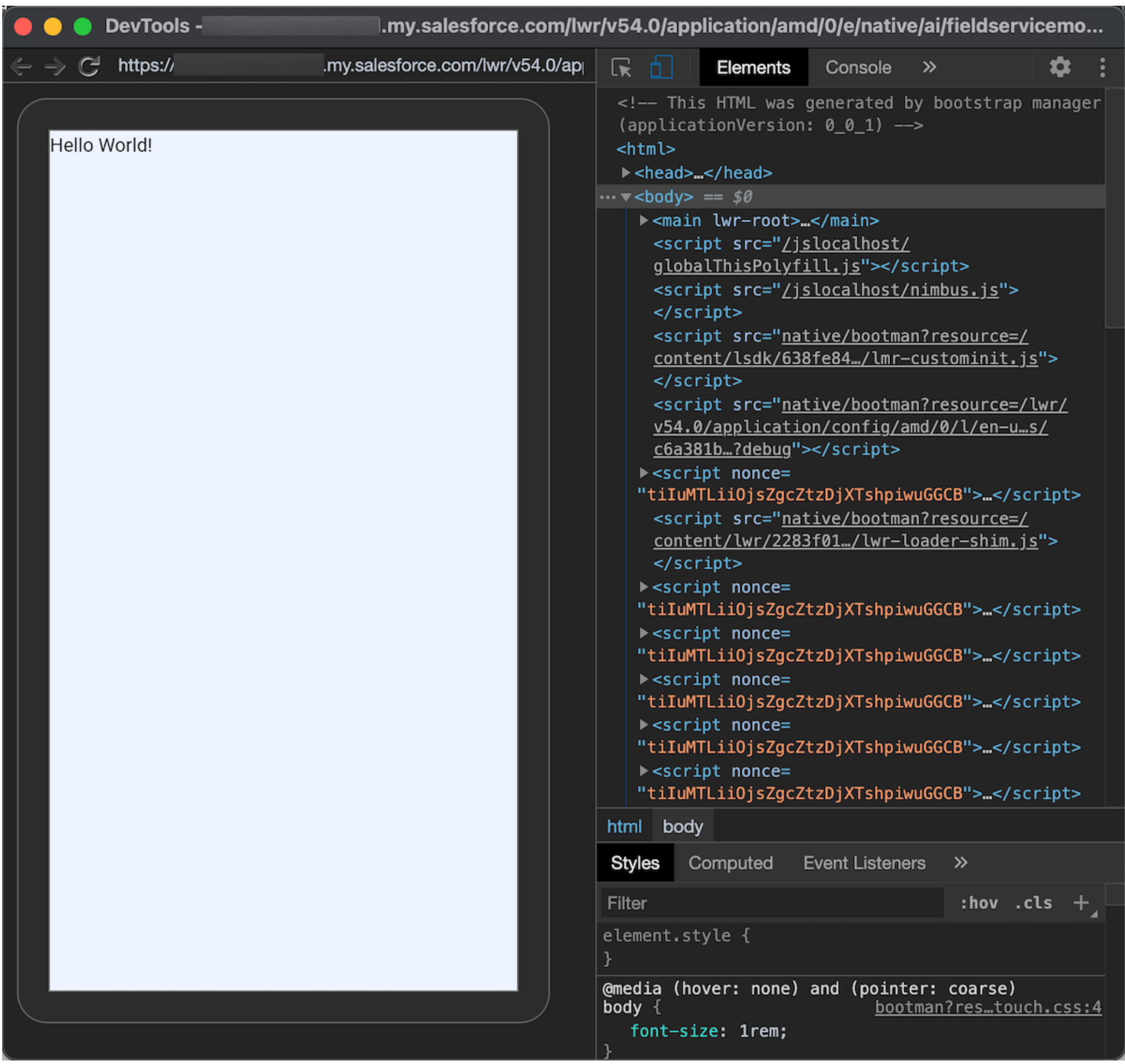
Debug in Android

Connect Chrome DevTools on your desktop to the WebView in the Field Service mobile app where your LWC is running.

To run the Field Service mobile app in an Android emulator, download and install the Salesforce Field Service mobile app APK file.

1. On your emulated Android device, open the Settings app.
2. Enter `About emulated device` into the search bar and click it.
3. Scroll to the bottom of the page and click **Build number** seven times.
   The message "You are now a developer!" appears when you click it enough, indicating that developer mode is enabled for the emulator.
4. Launch Chrome on your desktop.
5. In the location bar, enter `chrome://inspect/#devices`.
6. Click **Inspect** for the WebView under the Remote Target emulator you're using.

A window appears containing your emulator with Chrome DevTools connected to it. You can use Chrome DevTools to inspect the LWC element, set breakpoints, and see the console output. As with iOS, you can use the standard web development techniques for working with HTML, JavaScript, and CSS to run, test, debug, and improve your LWC.



See Remote debugging WebViews for additional basics, and Debugging embedded JavaScript in an Android app using Chrome DevTools for a deep dive on debugging JavaScript in an embedded WebView of an Android mobile app.