
Lightning Web Components Developer Guide

Version 44.0, Winter '19



Note: This release is in preview. Features described in this document don't become generally available until the latest general availability date that Salesforce announces for this release. Before then, and where features are noted as beta, pilot, or developer preview, we can't guarantee general availability within any particular time frame or at all. Make your purchase decisions only on the basis of generally available products and features.

 [@salesforcedocs](https://twitter.com/salesforcedocs)

Last updated: August 13, 2018

CONTENTS

Chapter 1: Introducing Lightning Web Components	1
Release Notes	2
Supported Browsers	3
Supported JavaScript	4
Get Set Up to Develop Lightning Web Components	4
Explore Lightning Web Components Sample Code	6
Create an Empty Project for Lightning Web Components	7
Lightning Web Components Available During Pilot	9
Chapter 2: Create Lightning Web Components	12
Define a Component	13
Create a Component Folder	13
Component HTML File	14
Component JavaScript File	14
Component Configuration File	15
Component CSS File	16
Additional JavaScript Files for Sharing Code	17
Component Tests	17
Component Namespaces	17
Create a Hello World Lightning Web Component	18
Component HTML Template Syntax	23
Render DOM Elements Conditionally	23
Render Lists	24
Use Getters Instead of Expressions	25
CSS	26
Style Components with Lightning Design System	26
Style Components with CSS	27
Component JavaScript Properties	29
JavaScript Property Names	30
Reactive Properties	30
Private Properties	37
Reflect JavaScript Properties to HTML Attributes	37
Compose Components	39
Data Binding Between Components	40
Use Slots as Placeholders	42
Shadow DOM	44
JavaScript Methods	46
Share JavaScript Code	49
Component Lifecycle	52

Contents

Component Lifecycle Hooks	52
Run Code When a Component Is Created	55
Run Code When a Component Is Inserted or Removed from the DOM	56
Run Code When a Component Renders	57
Handle Component Errors	59
Labels and Static Resources	60
Resources	60
Labels	62
Get Current User	63
Component Accessibility	64
Chapter 3: Communicate with Events	68
Simple Events	69
Events with Data	69
Extended Event Types	70
Creating an Event	71
Dispatching an Event	72
Handling an Event	72
Understanding Event Propagation	76
Sending Events to an Enclosing Aura Component	78
Chapter 4: Work with Salesforce Data	81
Use the Wire Service to Get Data	82
Wire Service Example: Get Record Data	86
User Interface API Wire Adapters and JavaScript Functions	88
Call Apex Methods	89
Chapter 5: Component Security with Locker Service	94
DOM Access Containment	95
Secure Wrappers for Global References	95
Restricted Access to Salesforce Global Variables	96
Unsupported Browser APIs During Pilot	96
Locker Service Disabled for Unsupported Browsers	97
Chapter 6: Use Lightning Web Components in Lightning Experience and Salesforce	98
View a List of Lightning Components in Your Org	99
Navigate to Pages in Lightning Experience and Salesforce	99
PageReference Types	106
Configure a Component for Lightning Pages and Lightning App Builder	110
Component Configuration File Tags	111
Make Your Component Width-Aware with lightning-flexipage-service	113
Chapter 7: Lightning Web Components and Aura Components Working Together	116
Component Naming Schemes	117

Understanding Aura Component Facets	117
Lightning Web Components Inherit Aura Component Styling	119
Share JavaScript Code in Lightning Web Components and Aura Components	120
Chapter 8: Migrate Aura Components to Lightning Web Components	122
Migration Strategy	123
Pick a Component to Migrate	123
Migrate Component Bundle Files	123
Migrate Markup	124
Migrate Attributes	125
Migrate Iterations	125
Migrate Conditionals	126
Migrate Expressions	126
Migrate Initializers	127
Migrate Facets	127
Migrate Base Components	128
Migrate Registered Events	129
Migrate Event Handlers	129
Migrate Events	129
Migrate CSS	130
Migrate JavaScript	130
Migrate Apex	131
Data Binding Behavior Differences With Aura	131
Chapter 9: Test Lightning Web Components	133
Tools for Testing Lightning Web Components	134
Unit Test Lightning Web Components with Jest	135
Install Jest for Testing Lightning Web Components	135
Write Jest Tests for Lightning Web Components	137
Run Jest Tests for Lightning Web Components	139
Chapter 10: Debugging	140
Enable Debug Mode	141
Chapter 11: Lightning Web Components Reference	142
Components	143
HTML Template Directives	145
Lifecycle Hooks	146
lightning-ui-api* Wire Adapters and Functions	147
Supported Salesforce Objects	148
lightning-ui-api-list-ui	149
lightning-ui-api-lookups	151
lightning-ui-api-object-info	152
lightning-ui-api-record	154
lightning-ui-api-record-ui	159

Contents

Validations for Lightning Web Component Code 160

Chapter 12: Glossary 165

INDEX 170

CHAPTER 1 Introducing Lightning Web Components

In this chapter ...

- [Release Notes](#)
- [Supported Browsers](#)
- [Supported JavaScript](#)
- [Get Set Up to Develop Lightning Web Components](#)
- [Explore Lightning Web Components Sample Code](#)
- [Create an Empty Project for Lightning Web Components](#)
- [Lightning Web Components Available During Pilot](#)

Now you can develop Lightning components in two models: the original model—Aura, and the new model—Lightning Web Components. The new model is based on web standards and built for performance—we designed it to be familiar and intuitive for developers. Use your favorite dev tools to create Lightning web components using HTML, JavaScript, and CSS.

Lightning web components and Aura components can work together in the same app. You can migrate incrementally, one component at a time.



Tip: The name of the development model is Lightning Web Components, uppercase. When we refer to the components themselves, we use lowercase, Lightning web components.

What Are Web Components?

The Lightning Web Components model is based on Web Components, which is a set of W3C web specifications.

- Custom Elements—Create your own reusable HTML tags using HTML, JavaScript, and CSS.
- HTML Template—Create templates that can be loaded dynamically at runtime.
- Shadow DOM—Encapsulates DOM and CSS within a custom element.
- HTML Imports—Define dependencies between HTML files.

Three of the [Web Components specs](#) are in draft status, and aren't natively implemented in every browser. The HTML Template spec is complete and is part of [HTML 5](#).

If you're new to Web Components, [this article](#) has a great introduction to all the moving parts.

What Are Lightning Web Components?

Lightning Web Components is an abstraction on top of the Web Components specs. In contrast to web components, Lightning web components work in all browsers supported by Salesforce.

Lightning Web Components supports the good parts of the Web Components specs—the parts that perform well in browsers. Automatic wiring between HTML, JavaScript, and CSS makes it easier to build Lightning web components.

Release Notes

Here are the latest and greatest Lightning Web Components updates in Winter '19.

Breaking Changes

These changes break your existing code and require that you update it to use the new syntax.

Composed Event Retargeting

When an event propagates to another component, the `target` value of the event is changed to the custom element (component) that dispatched the event. The event doesn't point at the tag within the custom element that dispatched the event. For example, if a `lightning-button` element inside a component dispatches an event, `event.target` in a parent component references the component that contains the `lightning-button`. This behavior is consistent with standard Shadow DOM behavior. See [Handling an Event](#).

Native `change` Events Are No Longer Composed

Before Winter '19, native change events were composed, which allowed them to escape their shadow root. In Winter '19, change events are not composed. Update event handlers that listen for a retargeted change event.

Navigation Service Changes

The `getPageReference` wire adapter is now called `currentPageReference`. The `getNavigationService` function is now called `NavigationMixin`. Instead of exporting the function and using it in a wire annotation, you now export `NavigationMixin` and extend the base class with your custom navigation class. See [Navigate to Pages in Lightning Experience and Salesforce](#).

`parentElement` Returns `null` On Shadow Root Children

Previously, the `parentElement` property on shadow root children would return the shadow root. This behavior is incorrect and now returns `null`. To access the shadow root from a child, use `parentNode` instead.

Slot Tags Render in HTML

Previously, `<slot>` elements were not rendered in HTML output. Now, consistent with native Shadow DOM slotting, they render in HTML.

Root Property on `Element` Renamed `template`

In alignment with recent Lightning Web Components engine changes, the `root` property on the `Element` instance has been renamed `template`.

Deprecated Function: `createRecordInputFromRecord`

This function has been replaced by [generateRecordInputForCreate](#) and [generateRecordInputForUpdate](#).

`querySelector` Scope

Using `element.querySelector` will only return elements that have been passed to your element via slots, the same behavior as using `this.querySelector` inside of a template. Using `element.template.querySelector` will only return elements that are defined inside of your template.

Accessing Global Salesforce Values

To access global Salesforce values for Apex, schema, resource, label, and user, follow the structure `@salesforce/apex`, `@salesforce/schema`, `@salesforce/resource-url`, `@salesforce/label`, and `@salesforce/user`. Note that these values cannot be imported by themselves; you must indicate the property that you want to import.

Non-Breaking Changes

These changes don't break existing code.

Import References to Salesforce Objects and Fields

If you're using a wire adapter in the `lightning-ui-api-*` namespace, we strongly recommend importing references to objects and fields. Salesforce verifies that the objects and fields exist, prevents objects and fields from being deleted, and cascades any renamed objects and fields into your component's source code. Importing references to objects and fields ensures that your code works, even when object and field names change. See [Use the Wire Service to Get Data](#).

Delete a Record

Use a new JavaScript API to delete a Salesforce record. See `deleteRecord(recordId)`.

Give a Component the ID of the Current Record

In the component's configuration file, set `<tag>` to the new value `lightning__HasRecordId`. In the component's JavaScript class, define a public `recordId` property. When the component is invoked in a record context in Lightning Experience or the Salesforce app, `recordId` is set to the ID of the record being viewed. See [Component Configuration File](#).

Client-Side Caching for Apex Method Results

To improve runtime performance, add `@AuraEnabled(cacheable=true)` to an Apex method to cache the method results on the client. To set `cacheable=true`, a method must only get data, it can't mutate data. See [Call Apex Methods](#).

Use `util` in Jest Tests To Access `ShadowRoot`

Test authors should now add `import { getShadowRoot } from 'lwc-test-utils'` to access `shadowRoot`.

Make Your Component Width-Aware with `lightning-flexipage-service`

`lightning-flexipage-service` provides page region information to the component that contains it. Use the `getRegionInfo` wire adapter to pass the width of the region where the component is dropped in the Lightning App Builder. See [Make Your Component Width-Aware with `lightning-flexipage-service`](#).

Reference Aura Design Tokens in Lightning Web Components

Design tokens are named entities that store visual design attributes, such as margins and spacing values, font sizes and families, or hex values for colors. Define design tokens in Aura components and use them in Lightning web components. See [Lightning Web Components Inherit Aura Component Styling](#).

New `render()` Method

For complex tasks like conditionally rendering a template or importing a custom one, use `render()` to override standard rendering functionality. This function gets invoked after `connectedCallback()` and must return a valid HTML template.

See [Component Lifecycle Hooks](#).

Additional Components to Streamline Your Code

For Winter '19, we've added the following components: **`lightning-record-view-form`**, **`lightning-record-edit-form`**, **`lightning-record-form`**, **`lightning-input-field`**, and **`lightning-output-field`**.

Supported Browsers

Lightning Web Components supports the browsers that Lightning Experience supports.

[Supported Browsers for Lightning Experience](#)

These versions of the supported browsers take full advantage of Lightning Web Components performance enhancements.

Browser Name	Version
Google Chrome™	59+
Microsoft® Edge	15+
Mozilla® Firefox®	54+

Browser Name	Version
Apple® Safari®	11.x+

For earlier versions of these browsers, and for IE 11, Lightning Web Components uses compatibility mode. Compatibility mode uses the lowest common denominator—code is transpiled down to ES5 and the required polyfills are added.

Lightning web components work correctly in compatibility mode, but they miss optimizations and don't perform as well.

Supported JavaScript

Take advantage of modern development practices, including the latest versions of JavaScript.

Lightning Web Components JavaScript support includes:

- ES7 (ECMAScript 2016)
- ES8 (ECMAScript 2017)—excluding [shared memory and atomics](#)
- ECMAScript 2018—only [object rest properties and object spread properties](#)

If you're not familiar with ECMAScript, or if you want to know more about how the JavaScript language is developed, the [2ality blog](#) provides a clear and concise explanation. Because of the feature proposal process described in the blog, the [ES7 release](#) and the [ES8 release](#) included only a few features.

If you haven't used JavaScript in awhile (or if you've never used it), start by learning ES6. ES6 was a huge upgrade to the language and includes many syntax changes. People on the web seem to agree that [Luke Hoban's GitHub repo](#) is a good place to start.

Get Set Up to Develop Lightning Web Components

To develop Lightning web components during the pilot, you must install and use Salesforce DX. To have fun developing Lightning web components, we recommend installing Visual Studio Code with Salesforce extensions.

Enable a Dev Hub Org for the Pilot

During the pilot, develop web components in scratch orgs that you create from a Dev Hub via the Salesforce CLI. We recommend that you use a dedicated Dev Hub org because all scratch orgs created during the pilot will be deleted when the pilot ends. Preserve your source code in your version control system, not in orgs used during the pilot.

Sign up for a Summer '18 (API 43.0) Dev Hub org.

- [Salesforce DX org sign-up](#)

 **Note:** Dev Hub is enabled by default in orgs created using the Salesforce DX org sign-up. For orgs created using other sign-up processes, [enable dev hub](#). During the pilot, you must use a Developer Edition org.

Enable My Domain

To use Lightning web components in your org, set up a My Domain name, which is a subdomain within the salesforce.com domain. For example, `trailhead` is a subdomain of the Salesforce domain: `trailhead.salesforce.com`.

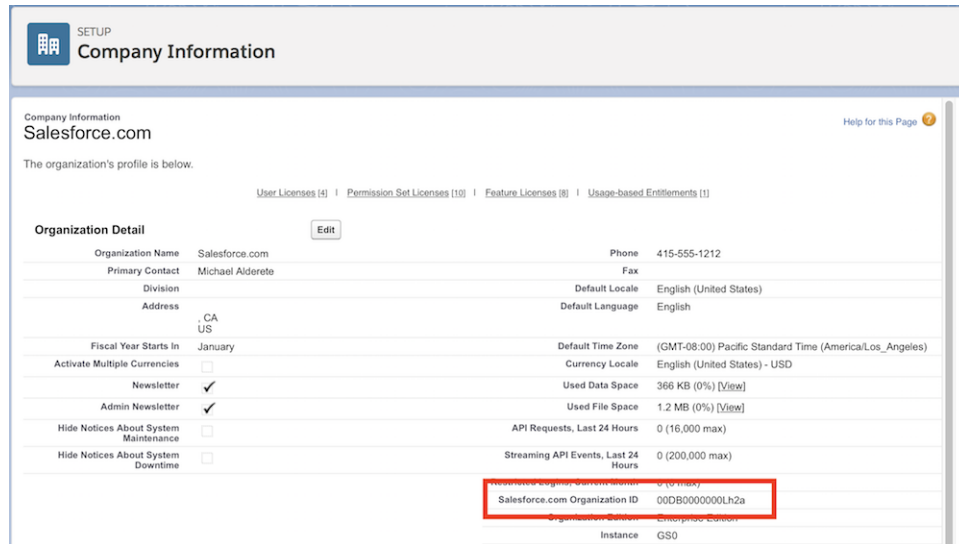
When My Domain isn't deployed in your org, user interface controls related to Lightning web components may be hidden or inactive. Lightning web components added to pages, tabs, and so on, don't run and may be omitted, or display a placeholder error message.

See [My Domain](#) in Salesforce Help.

Enable Lightning Web Components for Your Dev Hub Org

Send the Dev Hub org ID to the pilot coordinator via the [Lightning Web Components - Pilot](#) Chatter group.

To find your org ID, from Setup, enter *company* in the Quick Find box, then select **Company Information**.



Access the Samples Repos

If you don't have access to these samples repos, request it from the pilot coordinator via the [Lightning Web Components - Pilot](#) Chatter group.

- github.com/forcedotcom/sfdx-lwc-samples
- github.com/forcedotcom/ebikes-lwc

Install the Latest Version of Salesforce CLI

To work with Lightning web components using Salesforce DX, install the latest version of the Salesforce command-line interface (CLI). Use a version that supports API version 43.0 or later. [Install the Salesforce CLI](#), or update an already installed version with the following command.

```
sfdx plugins:update
```

Once you have the latest version installed, run this command to check the API version support.

```
sfdx plugins --core
```

The result include an entry and version for the `salesforcedx` plugin, something like `salesforcedx 43.0.5`.

Install Visual Studio Code

We really, *really* recommend installing these tools.

- [Visual Studio Code](#) (VS Code)

In combination with the next items, VS Code is the best source code editor for working with Lightning web components.

- [Salesforce Extensions for VS Code](#)

This extensions pack lets you work with Salesforce DX and Lightning Platform code directly within VS Code. It includes tools, code syntax highlighting, suggestions, and validation for various Salesforce platform languages.

- [LWC Code Editor for Visual Studio Code](#)

This separate extension includes preliminary support for Lightning Web Components. To learn about LWC Code Editor features, check out the [ReadMe](#).

The following tools, while not strictly essential, are also recommended and used by the engineering team behind Lightning Web Components.

- [ESLint Extension for VS Code](#)

Integrates the outstanding [ESLint](#) code validation ("linting") tool into VS Code. ESLint helps you avoid silly, and not-so-silly, JavaScript coding errors.

- [EditorConfig Extension for VS Code](#)

Integrates [EditorConfig](#) support into VS Code. Useful for ensuring consistent team coding styles.

Explore Lightning Web Components Sample Code

We've created GitHub repositories of Lightning Web Components sample code. Clone the repositories to your local machine and start playing in no time!

Clone the **sfdx-lwc-samples** Repo

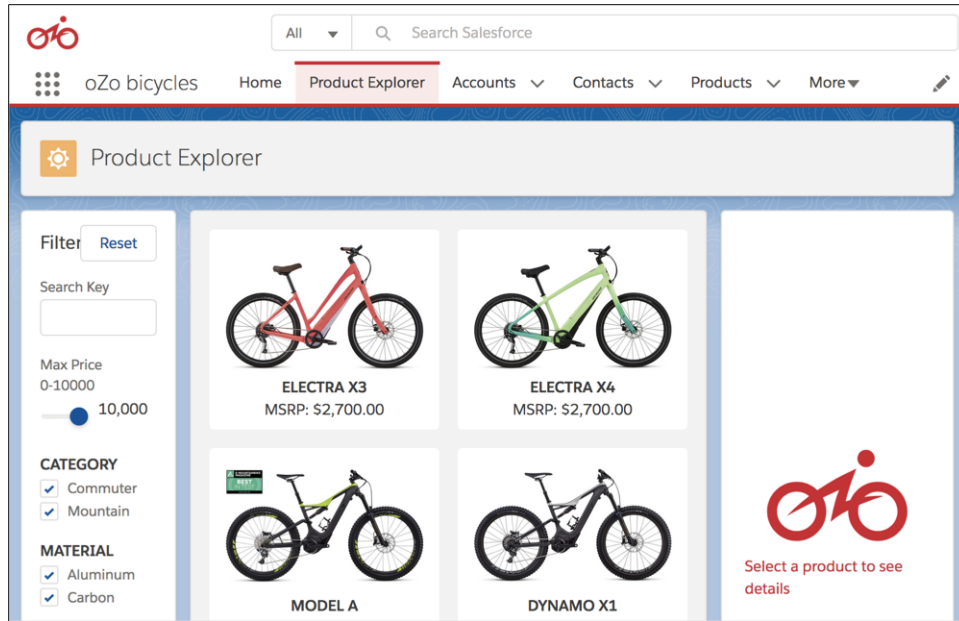
This samples repo contains an app called Lightning Web Components. The app is a tutorial that shows you how to program using the Lightning Web Components model. Step through it to build your first Lightning web component.

To clone the repo and get started, follow the directions in the [ReadMe](#).

After you clone the repo, you can also follow the steps to [Create a Hello World Lightning Web Component](#).

Clone the **ebikes-lwc** Repo

This repo contains an app called oZo bicycles. The app's Product Explorer was built using Lightning web components.



To clone the repo and get started, follow the directions in the [ReadMe](#).

Create an Empty Project for Lightning Web Components

Create your own Salesforce DX project to develop Lightning web components in a clean environment.

Create a Salesforce DX Project

A Salesforce DX project has a specific structure and a configuration file that identifies the directory as a Salesforce DX project.

Create Salesforce DX projects for all your Lightning web components pilot work.

```
cd path/to/your/sfdx/projects
sfdx force:project:create --projectname MyLightningWebComponents
cd MyLightningWebComponents
```

Important: Don't create a project within an existing project, that is, don't use the `force:project:create` command when your current directory is within a Salesforce DX project. The `sfdx-lwc-samples` and `ebikes-lwc` samples repos are both pre-configured as Salesforce DX projects.

Authenticate Your Pilot-Enabled Dev Hub Org

Before you can run any new code, authenticate your Dev Hub org. Running this command opens a browser to the Salesforce login page where you enter your Dev Hub username and password. Authenticate only once, not every time you work on your project.

In VS Code, press Command + Shift P, enter `sfdx`, and select **SFDX: Authorize a Dev Hub**.

You can also run this command from the command line.

```
sfdx force:auth:web:login -d -a LWC-Hub
```

"LWC-Hub" is an alias for the Dev Hub org that you can use in other Salesforce CLI commands.

Create a Pilot-Ready Scratch Org Definition File

Create a scratch org definition file with the following additions:

```
{
  "orgName": "Your Name Here",
  "edition": "Developer",
  "features": ["LightningWebComponents"],
  "orgPreferences" : {
    "enabled": ["S1DesktopEnabled"],
    "disabled": ["S1EncryptedStoragePref2"]
  }
}
```

The `LightningWebComponents` feature allows you to run and test your Lightning web components in scratch orgs you create. This feature can be enabled only for Developer Edition orgs during the pilot. The `S1DesktopEnabled` preference enables Lightning Experience in the org. Disabling the `S1EncryptedStoragePref2` preference prevents client-side caching of component definitions, which allows you to reload a changed component after a `force:source:push`.

Create a Lightning Web Components-Enabled Scratch Org

Use your scratch org definition file to create Lightning Web Component-enabled scratch orgs. This example uses a scratch org definition file named `lwc-pilot-scratch-def.json`, which is saved in the Salesforce DX standard `config` directory.

In VS Code, press Command + Shift P, enter `sfdx`, and select **SFDX: Create a Default Scratch Org**.

You can also run this command from the command line.

```
sfdx force:org:create -s -f config/lwc-pilot-scratch-def.json -a "WebComponentsPilot"
```

"WebComponentsPilot" is an alias for the scratch org that you can use in other Salesforce CLI commands.


Push Source to the Scratch Org

You're ready to push new source code, including Lightning web components, to your scratch org for testing. Save your code first!

In VS Code, press Command + Shift P, enter `sfdx`, and select **SFDX: Push Source to Default Scratch Org**.

You can also run this command from the command line.

```
sfdx force:source:push
```

 **Important:** For the Lightning web components pilot, the only `sfdx` command that's supported for Lightning web components is `force:source:push`. You can't use `force:source:pull`, `force:source:open`, use a `lightning:*:create` to create components from a template, or any other `sfdx` command. Use Visual Studio Code to create and edit component code, and `sfdx force:source:push` to push it to your scratch org.

Open the Scratch Org

In VS Code, press Command + Shift P, enter `sfdx`, and select **SFDX: Open Default Scratch Org**.

You can also run this command from the command line.

```
sfdx force:org:open
```

Development Workflow

As you develop, it's easy to see how changes look in your org. Just push source to the scratch org and refresh the browser.


SEE ALSO:

[Salesforce DX Developer Guide: How Salesforce DX Changes the Way You Work](#)

[Create a Hello World Lightning Web Component](#)

Lightning Web Components Available During Pilot

The pilot program introduces enough base Lightning web components for you to compose efficient, memorable user interfaces.

 **Important:** During the pilot release, not all base Lightning components are available as Lightning web components. If you try to use one that isn't available, your component will save but you'll receive a runtime error.

To look up a component's attributes, and to explore its look and feel, use the Component Library.

- <https://<myCustomSalesforceDomain>.lightning.force.com/componentReference/suite.app>
- <https://developer.salesforce.com/docs/component-library>

During the pilot release, Lightning web components aren't described in the Component Library, but their Aura equivalents are. Lightning web components use a slightly different naming convention, but—with a few exceptions listed below—they have the same attributes. To find the documentation for a Lightning web component, convert its name to the Aura naming convention. Lightning web components use hyphen delimiters in their names and attributes. Aura components use capital letters.

For example, to find documentation for the `lightning-button-icon` Lightning web component, look for the Aura `lightning:buttonIcon` component. Don't forget to convert attributes as well. For example, `<lightning:buttonIcon iconName="utility:down"/>` becomes `<lightning-button-icon icon-name="utility:down"></lightning-button-icon>` (Lightning web components require closing tags).

These Lightning web components are available during pilot. The bold components are new in Winter '19. If there are differences between the Lightning web component and the Aura component, they're listed beside the component name.

- **lightning-accordion**
- **lightning-accordion-section**
- lightning-avatar
- lightning-badge
- lightning-breadcrumb
- lightning-breadcrumbs
- lightning-button: Doesn't support `body` because the Lightning web component doesn't have a slot.
- lightning-button-icon
- lightning-button-icon-stateful
- **lightning-button-menu**
- lightning-button-stateful
- **lightning-card**
- lightning-carousel
- lightning-checkbox-group
- lightning-click-to-dial

- `lightning-combobox`
- `lightning-datatable`
- `lightning-dual-listbox`
- `lightning-dynamic-icon`
- `lightning-file-upload`
- **lightning-flexipage-service**—See [Make Your Component Width-Aware with lightning-flexipage-service](#).
- `lightning-formatted-address`
- `lightning-formatted-date-time`
- `lightning-formatted-email`
- `lightning-formatted-location`
- `lightning-formatted-name`
- `lightning-formatted-number`
- `lightning-formatted-phone`
- `lightning-formatted-rich-text`
- `lightning-formatted-text`
- `lightning-formatted-time`
- `lightning-formatted-url`
- `lightning-helptext`
- `lightning-icon`
- `lightning-input`
- `lightning-input-address`
- `lightning-input-field`
- `lightning-input-location`
- `lightning-input-name`
- `lightning-input-rich-text`: To add buttons to the toolbar of the rich text editor, use the `custom-buttons` attribute instead of the Aura `body` facet. Pass the buttons as an array of objects.
- **lightning-layout**
- **lightning-layout-item**
- **lightning-menu-item**
- `lightning-navigation`: See [Navigate to Pages in Lightning Experience and Salesforce](#).
- **lightning-notifications-library**
- `lightning-output-field`
- `lightning-overlay-library`
- `lightning-pill`: Use a slot instead of the Aura `media` attribute. The Lightning web component also has a `variant` attribute, which changes the pill type to be a clickable link (`'link'`) or not (`'plain'`).
- `lightning-pill-container`
- `lightning-progress-indicator`
- `lightning-radio-group`
- `lightning-record-edit-form`
- `lightning-record-form`

- `lightning-record-view-form`
- `lightning-relative-date-time`
- `lightning-slider`
- `lightning-spinner`
- `lightning-textarea`
- `lightning-tree`
- `lightning-tree-grid`
- `lightning-vertical-navigation-item`
- `lightning-vertical-navigation-item-badge`
- `lightning-vertical-navigation-item-icon`

CHAPTER 2 Create Lightning Web Components

In this chapter ...

- [Define a Component](#)
- [Create a Hello World Lightning Web Component](#)
- [Component HTML Template Syntax](#)
- [CSS](#)
- [Component JavaScript Properties](#)
- [Compose Components](#)
- [Shadow DOM](#)
- [JavaScript Methods](#)
- [Share JavaScript Code](#)
- [Component Lifecycle](#)
- [Labels and Static Resources](#)
- [Component Accessibility](#)

A Lightning web component is a reusable custom HTML element with its own API.

A Lightning web component is composed of an HTML template file, a JavaScript file, and a configuration file. The files must use the same name so the framework can autowire them. The HTML template is linked to properties in the JavaScript class. A component can optionally include a CSS file and additional JavaScript files.

Define a Component

A Lightning web component must include an HTML file, a JavaScript file, and a configuration file. It can optionally include a CSS file and additional JavaScript files.

Create a Component Folder

To create a component, first create a folder that bundles your component's files.

Component HTML File

Every component requires an HTML file with the root tag `<template>`.

Component JavaScript File

Every component must have a JavaScript file that defines the HTML element. The file is an ES6 module.

Component Configuration File

Every component must have a configuration file. The configuration file defines the metadata values for the component, including the design configuration for components intended for use in Lightning App Builder.

Component CSS File

A component can include a CSS file. Use standard CSS syntax to style Lightning web components.

Additional JavaScript Files for Sharing Code

In addition to the JavaScript file that creates the HTML element, a component's folder can contain other JavaScript files. Use these JavaScript files to share code.

Component Tests

To create Jest tests for a component, create a folder called `__tests__` at the top level of the component's folder. Save the tests inside the `__tests__` folder.

Component Namespaces

Every component is part of a namespace. If your organization hasn't set a namespace prefix, use the default namespace, `c`, when referencing Lightning web components that you've created.

Create a Component Folder

To create a component, first create a folder that bundles your component's files.

The folder and the component's HTML template file, the component's JavaScript class file, the component's configuration file, and an optional CSS file must have the same name, including capitalization and underscores.

```
my_component
|--my_component.html
|--my_component.js
|--my_component.js-meta.xml
|--my_component.css
```

The folder and its files must follow these naming rules.

- Must use lowercase letters
- Must begin with a letter
- Must contain only alphanumeric or underscore characters
- Must be unique in the namespace
- Can't include whitespace

- Can't end with an underscore
- Can't contain two consecutive underscores
- Can't contain a hyphen

You can't create a Lightning web component with a name that collides with a custom Aura component in the same namespace. For example, if you have an Aura component named `c:progressBar`, you can't create a Lightning web component named `c-progress_bar`.

SEE ALSO:

[Component Namespaces](#)

[Component Naming Schemes](#)

Component HTML File

Every component requires an HTML file with the root tag `<template>`.

The HTML file follows the naming convention `<component>.html`, such as `my_component.html`.

Create the HTML for a Lightning web component declaratively, within the `<template>` tag. The HTML template element contains your component's HTML.

```
<!-- my_component.html -->
<template>
  <!-- Replace comment with component HTML -->
</template>
```

Store markup that you may not want to display when the page loads, but that can be rendered at runtime. See [Render DOM Elements Conditionally](#).

Use simple HTML template syntax to render lists. See [Render Lists](#).

Create placeholders in the template that component consumers can replace with content. See [Use Slots as Placeholders](#)

Component JavaScript File

Every component must have a JavaScript file that defines the HTML element. The file is an ES6 module.

The JavaScript file declares:

- The component's public API via public properties and methods
- Private properties
- Event handlers

ES6 Modules

JavaScript files in Lightning Web Components are ES6 modules.

By default, everything declared in a module is local—it's scoped to the module. To allow other code to use a class, function, or variable declared in a module, use the `export` statement. To import a class, function, or variable declared in a module, use the `import` statement.

Now let's look at how `export` and `import` are used in the Lightning web component's JavaScript file.

File Structure

Here's the basic structure of the JavaScript file.

```
// my_component.js
import { Element } from 'engine';
export default class MyComponent extends Element {
  // your properties and methods here
}
```

The core module in Lightning Web Components is `engine`. The `import` statement imports `Element` from the `engine` module.

```
import { Element } from "engine";
```

`Element` is a custom wrapper of the standard HTML element.

Extend `Element` to create a JavaScript class for a Lightning web component.

```
export default class MyComponent extends Element {
  // your content here
}
```

The `export default` keywords export a `MyComponent` class for usage in other components.

The convention is for the class name to be Pascal Case, where the first letter of each word is capitalized. In our example, the class name is `MyComponent` for a `c-my_component` component, where `c` is the default namespace.

We covered the basic syntax of a JavaScript component class. We'll see later how to use JavaScript properties to produce dynamic content.

SEE ALSO:

[Component JavaScript Properties](#)

[ES6 In Depth: Modules: How Salesforce DX Changes the Way You Work](#)

Component Configuration File

Every component must have a configuration file. The configuration file defines the metadata values for the component, including the design configuration for components intended for use in Lightning App Builder.

The configuration file follows the naming convention `<component>.js-meta.xml`, such as `helloworld.js-meta.xml`.

Include the configuration file in your component's project folder, and push it to your org along with the other component files.



Tip: If you don't include a configuration file for your component, you'll get an error similar to the following when you push your changes.

```
Cannot find Lightning Component Bundle <component_name>
```

Here's a simple configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
  <apiVersion>42.0</apiVersion>
  <isExposed>false</isExposed>
</LightningComponentBundle>
```

Here is a simple Lightning web component configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
  <apiVersion>42.0</apiVersion>
  <isExposed>false</isExposed>
  <masterLabel>Best Component Ever</masterLabel>
  <description>This is a demo component.</description>
</LightningComponentBundle>
```

This more complex version of the configuration file makes its component available for all Lightning page types, but restricts support on record pages only for account, opportunity, and custom objects. The component has a different set of properties defined for record pages than for app and Home pages. The configuration file also gives the component a record ID context, via

`<tag>lightning__HasRecordId</tag>`, so that when the component renders on a record page, the ID of that record page is passed to the component.

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
  <apiVersion>42.0</apiVersion>
  <isExposed>false</isExposed>
  <masterLabel>Best Component Ever</masterLabel>
  <description>This is a demo component.</description>
  <tags>
    <tag>lightning__RecordHome</tag>
    <tag>lightning__AppHome</tag>
    <tag>lightning__Home</tag>
    <tag>lightning__HasRecordId</tag>
  </tags>
  <tagConfigs>
    <tagConfig tags="lightning__RecordHome">
      <property name="prop1" type="String" />
      <objects>
        <object>Account</object>
        <object>Opportunity</object>
        <object>CustomObject__c</object>
      </objects>
    </tagConfig>
    <tagConfig tags="lightning__AppHome, lightning__Home">
      <property name="prop2" type="Boolean" />
    </tagConfig>
  </tagConfigs>
</LightningComponentBundle>
```

SEE ALSO:

[Component Configuration File Tags](#)

Component CSS File

A component can include a CSS file. Use standard CSS syntax to style Lightning web components.

To style a component, create a style sheet in the component bundle with the same name as the component. If the component is called `my_component`, the style sheet is `my_component.css`. The style sheet is applied automatically.

SEE ALSO:

[Style Components with CSS](#)

Additional JavaScript Files for Sharing Code

In addition to the JavaScript file that creates the HTML element, a component's folder can contain other JavaScript files. Use these JavaScript files to share code.

These additional JavaScript files must be ES6 modules and must have names that are unique within the component's folder.

```
my_component
|--my_component.html
|--my_component.js
|--my_component.js-meta.xml
|--my_component.css
|--shared_code.js
|--more_shared_code.js
```

SEE ALSO:

[Share JavaScript Code](#)

Component Tests

To create Jest tests for a component, create a folder called `__tests__` at the top level of the component's folder. Save the tests inside the `__tests__` folder.

```
my_component
|--my_component.html
|--my_component.js
|--my_component.js-meta.xml
|--my_component.css
|--__tests__
    |--my_component.test.js
```

Jest runs all JavaScript files in the `__tests__` directory. Test files must have names that end in `.js`, and we recommend that tests end in `.test.js` or `-test.js`. You can have a single test file with all of your component tests, or you can have multiple files to organize and group related tests. Test files can be placed in sub folders.

SEE ALSO:

[Write Jest Tests for Lightning Web Components](#)

Component Namespaces

Every component is part of a namespace. If your organization hasn't set a namespace prefix, use the default namespace, `c`, when referencing Lightning web components that you've created.

 **Important:** In Winter '19, only the default namespace, `c`, is supported.

A Lightning web component can reference another Lightning web component by adding `<mynamespace-my_component>` in its markup. The hyphen character separates the namespace from the component name.

For example, the `my_component` Lightning web component can reference the `hello_world` component in the `c` namespace by adding `<c-hello_world></c-hello_world>` in its markup.

```
<!-- my_component.html -->
<template>
  <c-hello_world></c-hello_world>
</template>
```

SEE ALSO:

[Create a Component Folder](#)

[Compose Components](#)

Create a Hello World Lightning Web Component

Let's create a simple Hello World Lightning web component that displays text from a variable defined in its JavaScript file.

The following steps assume you've completed [Get Set Up to Develop Lightning Web Components](#).

Build the Lightning web component and add it to the Lightning Web Components samples app. Then push the source code to a scratch org to get some experience with the deployment process.

1. If you haven't cloned the `sfdx-lwc-samples` GitHub repo, do it now!

```
cd path/to/your/sfdx/projects
git clone https://github.com/forcedotcom/sfdx-lwc-samples.git
cd sfdx-lwc-samples
```

2. In the `sfdx-lwc-samples/force-app/main/default/lightningcomponents/` directory, create a directory named `hello_world`.

Add 3 files to the directory.

- `hello_world.html`
- `hello_world.js`
- `hello_world.js-meta.xml`

You can complete this task with one CLI command. In the `lightningcomponents` directory, enter the following in the CLI:

```
sfdx force:lightning:component:create --type web --componentname hello_world
```

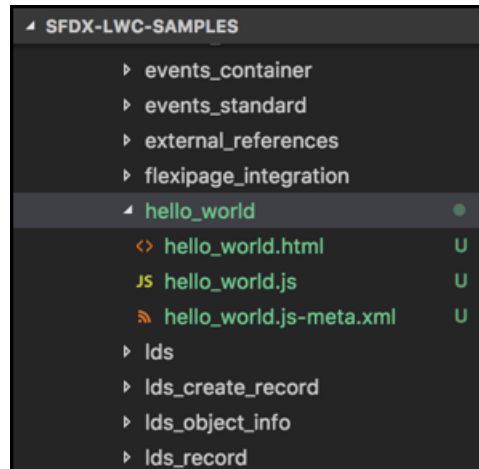
 **Note:** If you're using Salesforce DX version 44.0 or later, the `--type` flag requires the `lwc` value.

```
sfdx force:lightning:component:create --type lwc --componentname hello_world
```

To check your version of Salesforce DX, enter the following command in the CLI.

```
sfdx plugins --core
```


This image shows the files in Visual Studio Code.

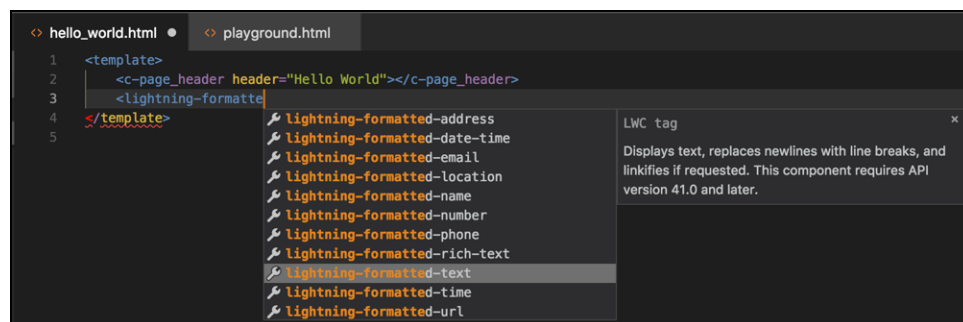


For more information about each file, see [Component HTML Template Syntax](#), [Component JavaScript File](#), and [Component Configuration File](#).

3. In the HTML file, add the `c-page_header` component and a `lightning-formatted-text` component, and save the file.

```
<template>
  <c-page_header header="Hello World"></c-page_header>
  <lightning-formatted-text value={greeting}
linkify="true"></lightning-formatted-text>
</template>
```

 **Tip:** Copying and pasting code from a PDF isn't fun, but using code completion in the [LWC Extension for Visual Studio Code](#) is!



The base Lightning components in this code can be found in the list of [Lightning Web Components Available During Pilot](#).

4. In the JavaScript file, import `Element`, which is a custom wrapper of the standard HTML element. Import it from `engine`, which is the core Lightning Web Components module. Define a class named `HelloWorld`, and define the variable `text`. Don't forget to save the file.


```
import { Element } from 'engine';
export default class HelloWorld extends Element {
  @api greeting;
}
```

If you copy and paste this example and your editor forces a line break, remove it.

You're almost done. The `hello_world.js-meta.xml` configuration file is required and defines some metadata for your component. Since the file is similar for all components, you can use the file you copied without editing it.

5. Open `playground.html` and add your new component. You can set the `greeting` attribute to any value you wish. If the value includes a URL or email address, they're converted to HTML `<a>` tags.

```
<template>
  Hello, Lightning Web Components!
  <hr>
  <c-hello_world greeting="HELLO FROM SALESFORCE.COM"></c-hello_world>
</template>
```

 **Note:** If you stepped through the `sfdx-lwc-samples` tutorial and completed the `playground` component, you see more code in the template. Add your `hello_world` component to the top of the template.

6. During the pilot, to display a Lightning web component in Lightning Experience, the component must be wrapped in an Aura component. The Aura wrapper component is just a container to expose your component in the UI. The samples repo contains Aura wrapper components that you can use as a template.

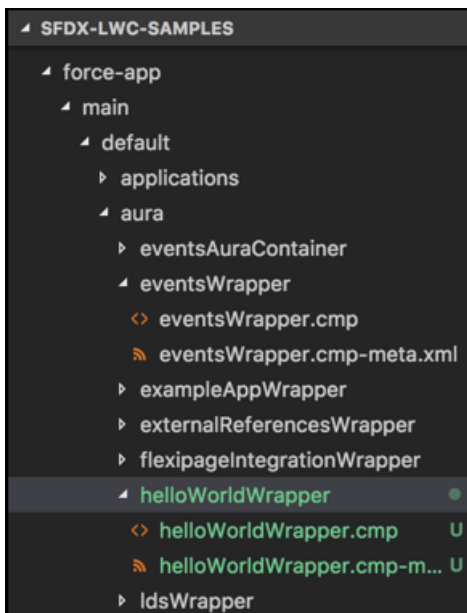
- a. Create a directory named `helloWorldWrapper` in `sfdx-lwc-samples/force-app/main/default/aura/`.

You can complete this task with one CLI command. In the `aura` directory, enter the following in the CLI:

```
sfdx force:lightning:component:create -n helloWorldWrapper
```

This command creates a bunch of files. The only ones you need are `helloWorldWrapper.cmp` and `helloWorldWrapper.cmp-meta.xml`.

This image shows the files in Visual Studio Code.



- b. The `helloWorldWrapper.cmp` Aura wrapper component only needs a reference to your `hello_world` component.

```
<aura:component implements="force:appHostable">
  <article class="slds-card">
    <c:hello_world/>
  </article>
</aura:component>
```

Don't edit `helloWorldWrapper.cmp-meta.xml`.

7. To add your Aura wrapper component to a tab in the Lightning Web Components samples app, define the tab. The tab uses the Aura wrapper component to display your component in the UI.

- a. In `sfdx-lwc-samples/force-app/main/tabs`, create the file `hello_world.tab-meta.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<CustomTab xmlns="http://soap.sforce.com/2006/04/metadata">
  <auraComponent>helloWorldWrapper</auraComponent>
  <label>Hello World</label>
  <mobileReady>false</mobileReady>
  <motif>Custom37: Bridge</motif>
</CustomTab>
```

- b. Set your tab's visibility. In `sfdx-lwc-samples/force-app/main/profiles/Admin.profile-meta.xml`, add an entry for your tab.

```
<tabVisibilities>
  <tab>hello_world</tab>
  <visibility>DefaultOn</visibility>
</tabVisibilities>
```

- c. Now, add your component to the tabs defined in the LWC app's configuration file `sfdx-lwc-samples/force-app/main/default/applications/LWC.app-meta.xml`.

```
<tabs>hello_world</tabs>
```

8. It's time to push your code to a scratch org. If you've already authenticated your Dev Hub org and created a scratch org for `sfdx-lwc-samples`, skip to step c.

- a. Authenticate your Dev Hub org.

In VS Code, press Command + Shift P, enter `sfdx`, and select **SFDX: Authorize a Dev Hub**. You can also run this command from the command line.

```
sfdx force:auth:web:login -d
```

- b. Create a scratch org.

In VS Code, press Command + Shift P, enter `sfdx`, and select **SFDX: Create a Default Scratch Org**. You can also run this command from the command line.

```
sfdx force:org:create -s -f config/project-scratch-def.json
```

Be patient, creating a scratch org can take a minute.

- c. From the `sfdx-lwc-samples` directory, push your project to the scratch org.

Save your source code files before you push! In VS Code, press Command + Shift P, enter `sfdx`, and select **SFDX: Push Source to Default Scratch Org**. You can also run this command from the command line.

```
sfdx force:source:push
```

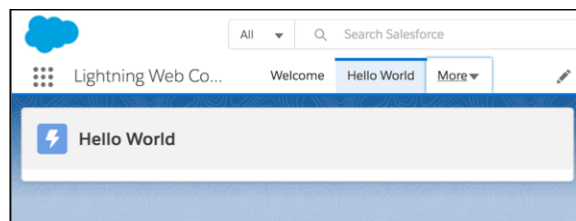
9. Open the app in Lightning Experience.

In VS Code, press Command + Shift P, enter `sfdx`, and select **SFDX: Open Default Scratch Org**. You can also run this command from the command line.

```
sfdx force:org:open
```

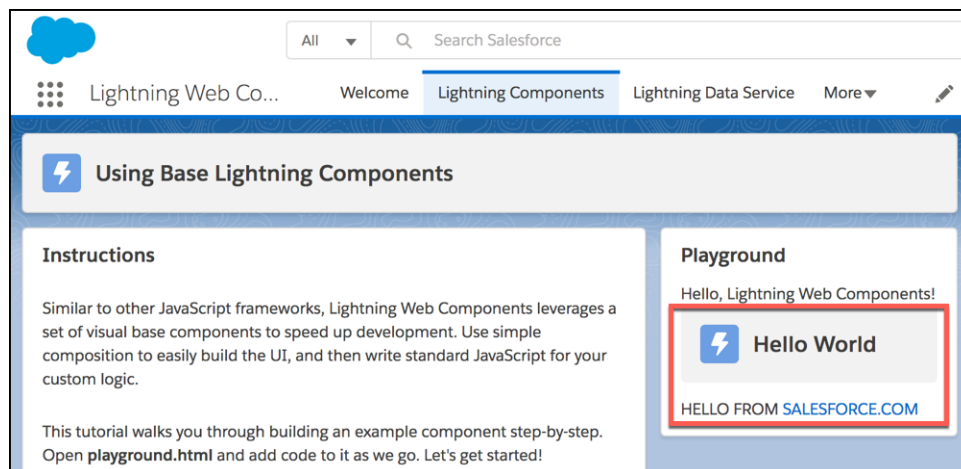
Next time you push changes, you can simply refresh the browser.

In the App Launcher, click the Lightning Web Components app and click Hello World.



This screenshot shows the component on its own without its `greeting` attribute set.

To see the component with its `greeting` attribute set, click the Lightning Components tab. The content inside the red square is the `hello_world` component nested in the `playground` component.



That's it!

What's next? Play with the samples! Add more `lightning` components to the `hello_world` component, check out the other sample components, build your own components, and have fun. As you experiment, use the Developer Guide to learn more about how to code Lightning web components.



Tip: Don't forget to clone the [ebikes-lwc](#) samples repo. It has a super cool Product Explorer built with Lightning web components.

Component HTML Template Syntax

Use HTML templates to render data to the DOM.

The `<template>` tag is a standard HTML tag that allows you to declare fragments of markup that are parsed as HTML.

Let's look at a simple web component with static text.

```
<template>
  <div class="slds-text-body_regular">Hello, World!</div>
</template>
```

A component with static text isn't very reusable. Use HTML directives and JavaScript to render content dynamically.

Render DOM Elements Conditionally

Use special HTML attributes called directives to render HTML content dynamically. To conditionally remove and insert DOM elements, use the `if:true|false={property}` directive.

Render Lists

To render a list of items from an array, use the `for:each` directive in a `<template>` tag and enclose the element to repeat.

Use Getters Instead of Expressions

Lightning Web Components doesn't support JavaScript or an expression language inside an HTML template. Instead, use a JavaScript getter to return the value of a property or a value computed in the JavaScript class.

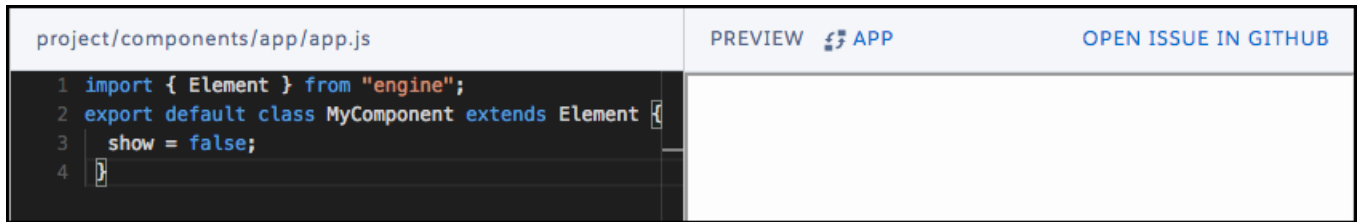
Render DOM Elements Conditionally

Use special HTML attributes called directives to render HTML content dynamically. To conditionally remove and insert DOM elements, use the `if:true|false={property}` directive.

In this example, the `if:true` and `if:false` directives display the `<p>` elements based on the truthiness and falsiness of the property `show` from the component's instance.

```
<template>
  <p if:true={show}>This content is visible.</p>
</template>
```

project/components/app/app.js	PREVIEW  APP	OPEN ISSUE IN GITHUB
<pre>1 import { Element } from "engine"; 2 export default class MyComponent extends Element { 3 show = true; 4 }</pre>	This content is visible.	



SEE ALSO:

[HTML Template Directives](#)

Render Lists

To render a list of items from an array, use the `for:each` directive in a `<template>` tag and enclose the element to repeat.

This example renders a list of California state parks.

The `for:each` directive takes an Array. In this example, it's an array of park names.

In the `for:item` directive, specify a new identifier to access the current item. This example uses `park`.


To access the item's zero-based index, use `for:index="index"` and specify a new identifier.

To improve rendering performance, you must use the `key={uniqueId}` directive to assign a unique identifier for each item in a list.

```
<!-- list.html -->
<template>
  <ul>
    <template for:each={parks} for:item="park" for:index="index">
      <li key={park.id}>{park.id}-{park.name}</li>
    </template>
  </ul>
</template>
```

```
// list.js
import { Element } from 'engine';
export default class List extends Element {
  parks = [
    { name: 'Lassen Volcanic National Park', id: '001' },
    { name: 'Channel Islands National Park', id: '002' },
    { name: 'Pinnacles National Park', id: '003' },
  ];
}
```

When a list changes, the framework uses the key to identify each item so that it can rerender only the item that changed. The `key` must be a string or a number, it can't be an object. You can't use `index` as a value for `key`. Assign unique keys to an incoming data set. To add new items, the component can track a private key-counter.

 **Important:** Using `key` is required, so don't forget your keys!

If you want to apply a special behavior to the first or last item in a list, use the `iterator` directive, which allows you to access the boolean properties `first` and `last`. (You can also access `index` and `value`.) The `iterator` directive must be used on a `template` tag.

```
<template>
  <template class="my-list" iterator:it={items}>
    <h1 if:true={it.first}>{it.value}</h1>
    <p if:false={it.first}>{it.value}</p>
  </template>
</template>
```

```
import { Element } from 'engine';

export default class App extends Element {
  items = [1, 2, 3, 5];
}
```

 **Note:** Lightning Web Components doesn't support iterating through object properties.

SEE ALSO:

[HTML Template Directives](#)

Use Getters Instead of Expressions

Lightning Web Components doesn't support JavaScript or an expression language inside an HTML template. Instead, use a JavaScript getter to return the value of a property or a value computed in the JavaScript class.

Getters replace expressions in Aura. In fact, getters are much more powerful than expressions because they're JavaScript functions. Getters also enable unit testing, which reduces bugs and increases fun.

1. Define a getter that computes the value in your JavaScript class.
2. Access the getter from the template using the `{getterOrPropertyName}` syntax.

The standard JavaScript `get` syntax binds an object property to a function that is called when that property is looked up.

```
get prop() { ... }
```


This JavaScript class contains a `hasSpaces()` getter, which returns whether the `lastName` private property includes any spaces.

```
import { Element } from 'engine';
export default class Getter extends Element {
  lastName = "Batman";

  get hasSpaces() {
    if (this.lastName && this.lastName.indexOf(' ') > -1) {
      return 'Yes';
    }
    return 'No';
  }
}
```

The component's markup returns the value of the `hasSpaces()` getter.

```
<template>
  <p>
    Does the name have spaces? {hasSpaces}
  </p>
</template>
```

 **Note:** Don't add extra spaces around the property in curly braces. The text inside the curly braces must be a valid JavaScript identifier or a member expression. For example `{data}` or `{data.name}` is valid, while `{ data }` is invalid because of the extra spaces.

SEE ALSO:

[Component JavaScript Properties](#)

CSS

To style components, use Lightning Design System or CSS.

To give your component the Lightning Experience look and feel, use Lightning Design System. To go your own way, use CSS.

[Style Components with Lightning Design System](#)

Salesforce Lightning Design System is a CSS framework that provides a look and feel that's consistent with Lightning Experience. Use Lightning Design System styles to give your custom Lightning web components a UI that is consistent with Salesforce, without having to reverse-engineer our styles. And best of all, it just works with Lightning components running in Lightning Experience and in the Salesforce mobile application.

[Style Components with CSS](#)

Use standard CSS syntax to style Lightning web components. Because Lightning web components respect Shadow DOM encapsulation, styles apply only to the component you create and don't impact other components on the page. Encapsulation means that components are self-contained and can be reused in different contexts.

SEE ALSO:

[Lightning Web Components Inherit Aura Component Styling](#)

Style Components with Lightning Design System

Salesforce Lightning Design System is a CSS framework that provides a look and feel that's consistent with Lightning Experience. Use Lightning Design System styles to give your custom Lightning web components a UI that is consistent with Salesforce, without having to reverse-engineer our styles. And best of all, it just works with Lightning components running in Lightning Experience and in the Salesforce mobile application.

Components in the `lightning` namespace already use [Lightning Design System](#). They also interoperate with Lightning web components. For example, this template is the `lightning:tree` component. It's easy to spot the Lightning Design System styles because they begin with `slds`.

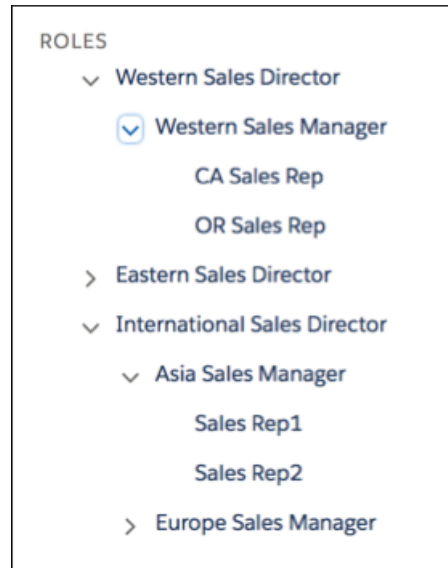
```
<template>
  <div class="slds-tree_container" role="application">
    <h4 class="slds-text-title_caps" id="treeheading">{header}</h4>
    <template if:true={state.node}>
```



```

        <lightning-tree-item class="slds-tree" role="tree" aria-labelledby="treeheading"
node={state.node} is-root="true">
            </lightning-tree-item>
        </template>
    </div>
</template>

```



Custom components that live in Lightning Experience or in the Salesforce mobile application can use Lightning Design System without any import statements or static resources. Assign a Lightning Design System CSS class to an HTML element, and you're good to go.

To build a Lightning web component, compose it by combining smaller `lightning` base components into a more complex, custom component. Try to construct your design from components like buttons and media objects. Use utilities, such as grid and spacing classes for layout. When you add new functionality to the component, search [Lightning Design System](#) for patterns that work and apply the classes to the template.

Style Components with CSS

Use standard CSS syntax to style Lightning web components. Because Lightning web components respect Shadow DOM encapsulation, styles apply only to the component you create and don't impact other components on the page. Encapsulation means that components are self-contained and can be reused in different contexts.

Create a Style Sheet

To style a component, create a style sheet in the component bundle with the same name as the component. The style sheet is applied automatically.

In this example, the template `todo-app.html` imports its associated style sheet, `todo-app.css`.

```

<!-- todo-app.html -->
<template>
    <h1>To-Do List</h1>
    <div class="listing">

```

```

        <template for:each={state.list} for:item="item">
            <c-todo-item label={item.text}></c-todo-item>
        </template>
    </div>
</template>

```

```

/* todo-app.css */
h1 {
    font-size: 2rem;
}

c-todo-item {
    border-bottom: 1px solid #eee;
}

c-todo-item:last-child {
    border-bottom: 0;
}

```

Style sheets use standard CSS syntax. You can use all the standard selectors, rules, and media queries. In addition, styles are scoped to the component. Styles defined in your component don't interfere with elements outside your component. In this example, the `h1` and `ns-todo-item` selectors match only the elements that `todo-app.html` generates.

Each component can have only one style sheet. Components can't share style sheets.

Style the Host Element

Because Lightning web components have a Shadow DOM, you can use the standard `:host` selector to style the host element. CSS styles applied to the host element don't affect child components or parent components.

To style the host element, use the `:host` selector.

```

:host {
    display: block;
    background-color: palevioletred;
}

```

The `:host` selector accepts an optional list of selectors. To match, the host element must have a class that matches the passed selector.

```

/* Match host element if it has "active" class */
:host(.active) {
    background-color: rebeccapurple;
}

/* Match host element if it has "foo" or "bar" class */
:host(.foo, .bar) {
    background-color: firebrick;
}

```

Sometimes it's useful to style a component when it's used in a specific context. The `:host-context` selector accepts a list of selectors. For `:host-context` to apply, one of the passed selectors must match a component ancestor.

For example, you have a hello world component with default font color as dark gray (`#97ffff`). The following example adjusts the component's text color to white (`#ffffff`) when the `.dark-theme` color scheme is applied to the component's context so you can

see it with a dark background. Using this technique, your components can integrate into the styling of their host context (such as a document or other component).

```
/* hello_world.css */
:host-context(.dark-theme) {
  color: darkslategray;
  font-color: #ffffff
}
font-color: #97ffff;

/* hello_world.html */
<template>
  <c-page_header header="Hello World"></c-page_header>
  <br>
  <lightning-formatted-text value="Welcome to Lightning Web
Components!"></lightning-formatted-text>
</template>
```

CSS Support and Performance Impact

The style encapsulation attempts to match the [CSS Scoping Module Level 1 standard](#), with a few exceptions.

- No support for the `::slotted` pseudo-element.
- No support for the `>>>` deep combinator selector.

Scoped CSS affects performance, so use it with care. Each selector chain is scoped, and each compound expression passed to `:host()` and `:host-context()` is spread into multiple selectors. This transformation increases the size and complexity of the generated CSS. These increases mean more bits on the wire, longer parsing time, and longer style recalculation time. To ensure CSS encapsulation, each element adds an extra attribute, which also increases rendering time.

SEE ALSO:

[Lightning Web Components Inherit Aura Component Styling](#)

Component JavaScript Properties

Declare properties in your component's JavaScript class. Reference the properties in your component's template to dynamically update content.

Property and *attribute* are almost interchangeable terms and can be confusing. Generally speaking, in HTML we talk about attributes, and in JavaScript we talk about properties.

Beyond the Basics

If you've developed an Aura component, you're familiar with the term *attribute*. In a Lightning web component, the closest thing to an Aura attribute is a JavaScript property.

In Lightning web components, properties in JavaScript use camel-case while attributes in markup are in kebab-case (dash-separated) to match HTML standards.

JavaScript Property Names

Property names in JavaScript are in camel-case while HTML attribute names are in kebab-case (dash-separated) to match HTML standards. For example, a JavaScript property named `itemName` maps to an HTML attribute named `item-name`.

Reactive Properties

If the value of a reactive property changes, the component's template rerenders. A reactive property can be public or private.

Private Properties

A private property can be used only by the JavaScript class that defines it. Don't use a private property in a template as private properties aren't reactive and don't rerender on change.

Reflect JavaScript Properties to HTML Attributes

You can control whether public JavaScript properties appear as attributes in the rendered HTML of a Lightning web component. Allowing properties to appear as attributes is especially important when creating accessible components, because screen readers and other assistive technologies use HTML attributes.

JavaScript Property Names

Property names in JavaScript are in camel-case while HTML attribute names are in kebab-case (dash-separated) to match HTML standards. For example, a JavaScript property named `itemName` maps to an HTML attribute named `item-name`.

Don't start a property name with these characters.

- `on` (for example, `onClick`)
- `aria` (for example, `ariaDescribedby`)
- `data` (for example, `dataProperty`)

Don't use these reserved words for property names.

- `slot`
- `part`
- `is`
- [Global HTML attributes](#), which are attributes, such as `class` and `title`, that are common to all HTML elements.

Reactive Properties

If the value of a reactive property changes, the component's template rerenders. A reactive property can be public or private.

Public Reactive Properties

Public properties define the public API for a component. A parent component that uses the component in its markup can access the component's public properties. Public properties are reactive. If the value of a reactive property changes, the component's template rerenders any content that references the property.

Private Reactive Properties

To observe a component's internal state and rerender when its state changes, decorate a property with `@track`. If the value of a tracked property changes and it's referenced in the template, the component rerenders. Tracked properties are also called private reactive properties.

Reactive Property Data Types

There are some limitations on the depth of changes tracked for rerendering in reactive properties. The tracking depth depends on the type of the reactive property.

Boolean Properties

Boolean attributes on standard HTML Elements are set to `true` by adding the attribute to the element. The absence of the attribute defaults the attribute to `false`. Therefore, the default value of an attribute is always `false`. Lightning web components use the same principle for boolean properties.

Public Reactive Properties

Public properties define the public API for a component. A parent component that uses the component in its markup can access the component's public properties. Public properties are reactive. If the value of a reactive property changes, the component's template rerenders any content that references the property.

To mark a property as public, annotate it with the `@api` decorator.

When you use the `@api` decorator, you must import it explicitly from `engine`. For example:

```
import { Element, api } from "engine";
```



Note: Use [decorators](#) to add functionality to a property. A property can have only one decorator. For example, a property can't have `@api` and `@track` (private reactive property) decorators.

Here's an example of a `TodoItem` class with an `itemName` public property. This class is part of an `c-todoitem` component, where `c` is the namespace.

```
/* todoitem.js */
import { Element, api } from "engine";
export default class TodoItem extends Element {
  @api itemName;
}
```

Here's the component's template.

```
<!-- todoitem.html -->
<template>
  <div class="view">
    <label>{itemName}</label>
  </div>
</template>
```

When you use the `c-todoitem` component in another component, you can set the `itemName` property. Property names in JavaScript are in camel-case while HTML attribute names are in kebab-case (dash-separated) to match HTML standards. In `todoapp.html`, the `item-name` attribute in markup maps to the `itemName` JavaScript property of `c-todoitem`.

```
<!-- todoapp.html -->
<template>
  <div class="listing">
    <c-todoitem item-name="Milk"></c-todoitem>
    <c-todoitem item-name="Bread"></c-todoitem>
  </div>
</template>
```



Note: A component that declares a public reactive property can't set the property value, except at component construction time. A parent component that uses the component in its markup can set the component's public property value. In our example, the `c-todoitem` component can't update the value of the `itemName` property in the `todoitem.js` file.

Create Getters and Setters

To execute logic each time a public property is set, write a custom setter. If you write a setter for a public property, you must also write a getter.

Here's an example of a setter that changes the item name to uppercase. To rerender the custom element when the setter is called, create a private reactive property. In this example, the private reactive property is `itemNameUpper`. This property is set in the setter and returned by the getter.

```
import { Element, api, track } from 'engine';
export default class TodoItem extends Element {
  @track itemNameUpper;

  @api
  set itemName(value) {
    this.itemNameUpper = value.toUpperCase();
  }

  @api
  get itemName() {
    return this.itemNameUpper;
  }
}
```

If you declare a getter and a setter with `@api` decorators, don't declare the public property explicitly.

For example, when we added the getter and setter, we removed the explicit declaration of the public property. We removed this line:

```
@api itemName;
```

Getter and setter names are case-sensitive. If you create a getter or setter for an HTML property, the case of the getter and setter must match the case of the HTML property. For example, a setter for the HTML property `maxLength` also must be `maxLength`. It can't be `maxlength`.

SEE ALSO:

[JavaScript Property Names](#)

[Private Reactive Properties](#)

Private Reactive Properties

To observe a component's internal state and rerender when its state changes, decorate a property with `@track`. If the value of a tracked property changes and it's referenced in the template, the component rerenders. Tracked properties are also called private reactive properties.

The tracked property can be referenced directly in the template, or it can be used indirectly in a getter and setter.

When you use the `@track` decorator, import it explicitly from `engine`. For example:

```
import { Element, track } from 'engine';
```

You can use `@track` to decorate a property only; you can't use it to decorate an object. It's possible to decorate multiple properties with `@track`. A property can have only one decorator. For example, a property can't have both `@track` and `@api`.

Let's look at some sample code to see how `@track` works.

We have a component called `child`, with a public property called `itemName`. When a parent component sets the value of `itemName`, the `child.js` code changes the `itemName` to uppercase. (The component could perform any operation, this one is simple and useful for examples.)

To perform an operation, use a getter and setter. Within the getter and setter, use a tracked property to hold the computed value and cause the component to rerender when the value changes. This component uses `@track itemNameUpper`. We decorate `itemNameUpper` with `@track` so that the component rerenders when its value changes.

```
<!-- child.html -->
<template>
  <div>Child value: {itemName}</div>
</template>
```

```
// child.js
import { Element, api, track } from 'engine';
export default class Child extends Element {
  @track itemNameUpper;
  @api
  get itemName() {
    return this.itemNameUpper;
  }

  set itemName(value) {
    this.itemNameUpper = value.toUpperCase();
  }
}
```

In this example, the parent component consumes the `child` component. When you click the parent component, it updates a tracked `counter` property, which causes the component to rerender.

```
<!-- parent.html -->
<template>
  <div onclick={handleClick} class="listing">
    <div>
      Click to increment.<br>
      Expected: parent and child values update<br>
      {counter}
    </div>
    <br>
    <div>Parent changingGetter: {changingGetter}</div>
    <div>Parent changingVar: {changingVar}</div>
    <br>
    <c-child item-name={changingGetter}></c-child>
    <c-child item-name={changingVar}></c-child>
  </div>
</template>
```

```
//parent.js
import { Element, api, track } from 'engine';
export default class App extends Element {
  @track counter = 1;
  changingVar = 'initial changingVar';

  get changingGetter() {
    return this.counter + ' changingGetter';
  }
}
```

```

    }

    handleClick() {
        this.counter += 1;
        this.changingVar = this.counter + ' changingVar';
    }
}

```

To see how tracked properties cause components to rerender, let's play with the code.

In `parent.js`, remove `@track` from the `counter` property. Run the code and click the `parent` component. The `parent` component doesn't rerender. The `child` component doesn't rerender.

Put the `@track` decorator back on the `counter` property.

In `child.js`, comment out the line `// @track itemNameUpper`. Run the code and click the `parent` component. The `parent` component rerenders, but the `child` component doesn't rerender so the `child` component values don't update.



Note: You can't create a getter or setter for a private reactive property. Since the property is private, you can access or modify it directly in the code so a getter or setter isn't needed.

Reactive Property Data Types

There are some limitations on the depth of changes tracked for rerendering in reactive properties. The tracking depth depends on the type of the reactive property.

Lightning Web Components tracks changes to the internal values of these types of reactive properties:

- Primitive values
- Plain objects created with `{...}`
- Arrays created with `[]`

This behavior is subtle so let's look at some code. This class has a reactive property, `x`, which is an object.

```

import { Element, track } from 'engine';
export default class TrackObject extends Element {
    @track x = {
        a: "",
        b: ""
    };

    init() {
        this.x.a = "a";
        this.x.b = "b";
    }

    update() {
        this.x.a = "aa";
        this.x.b = "bb";
    }
}

```

The template has a few buttons that change the internal state of `x`. The `onclick` handlers for the buttons are wired to the `init()` and `update()` methods in the JavaScript file.

```

<template>
    <p>object prop: {x.a}</p>

```



```

    <p>object prop: {x.b}</p>

    <button onclick={init}>Init</button>
    <button onclick={update}>Update</button>
  </template>

```

When you click either button, the updated values of the member values of `x` are rerendered. Changes to `x.a` and `x.b` are tracked because `x` is a plain object.

Now, let's look at a similar component with a reactive property, `x`, of type `Date`.

```

import { Element, track } from 'engine';
export default class TrackDate extends Element {
  @track x;

  initDate() {
    this.x = new Date();
  }

  updateDate() {
    this.x.setHours(7);
  }
}

```

Similarly to our previous example, the template has a few buttons that change the internal state of `x`.

```

<template>
  <p>Date: {x}</p>

  <button onclick={initDate}>Init</button>
  <button onclick={updateDate}>Update</button>
</template>

```

When you click the **Init** button, the change is tracked and the template is rerendered. Lightning Web Components can track that `x` is pointing to a new `Date` object. However, when you click **Update**, the template is not rerendered. Lightning Web Components doesn't track changes to the value of the `Date` object.



Note: When you set a reactive property to a value that can't be tracked, a warning is logged. If you're trying to debug a scenario where the component isn't rerendering on change, look in your browser console. For our example, the browser console logs this helpful warning:

```

Property "x" of [object:vm TrackDate] is set to a non-trackable object,
which means changes into that object cannot be observed.

```

Boolean Properties

Boolean attributes on standard HTML Elements are set to `true` by adding the attribute to the element. The absence of the attribute defaults the attribute to `false`. Therefore, the default value of an attribute is always `false`. Lightning web components use the same principle for boolean properties.

Statically Setting a Property

If you want to toggle the value of a boolean property in markup, you must default the value to `false`. For example:

```
/* bool.js */
import { Element } from 'engine';

export default class Bool extends Element {
  // Always set the default value for a boolean to false
  @api show = false;
}
```

Here's the HTML file.

```
<!-- bool.html -->
<template>
  <p>show value: {show}</p>
</template>
```

This parent component includes `c-bool`, which displays the default value of `false` because the `show` attribute isn't added to `<c-bool>`.

```
<!-- parent.html -->
<template>
  <c-bool></c-bool>
</template>
```

To set the `show` property to `true`, add a `show` attribute with an empty value to the markup. This version of `c-parent` displays a value for `true` for the `show` property.

```
<!-- parent.html -->
<template>
  <c-bool show></c-bool>
</template>
```

If you set the default value for the `show` property to be `true` instead in `bool.js`, there's no way to statically toggle the value to `false` in markup.

Dynamically Setting a Property

To toggle the value if the default property value is `true`, you can pass down a dynamic computed value from the parent component. For example:

```
<!-- parent.html -->
<template>
  <c-bool show={computedValue}></c-bool>
</template>
```

Use a JavaScript getter in `parent.js` to return the value of `{computedValue}`.

SEE ALSO:

[Use Getters Instead of Expressions](#)

Private Properties

A private property can be used only by the JavaScript class that defines it. Don't use a private property in a template as private properties aren't reactive and don't rerender on change.

This example declares a private `itemId` property. A private property doesn't have any decorator.

```
import { Element, api } from 'engine';
export default class TodoItem extends Element {
  @api itemTitle = ''; // public
  itemId = '';         // private
}
```

Private properties aren't reactive. If you want to use an internal property in the template, make it a reactive property instead by adding the `@track` decorator.

```
import { Element, api, track } from 'engine';
export default class TodoItem extends Element {
  @api itemTitle = ''; // public
  itemId = '';         // private
  @track state = { x: 100, y: 100 }; // internal and reactive
}
```

SEE ALSO:

[Reactive Properties](#)

Reflect JavaScript Properties to HTML Attributes

You can control whether public JavaScript properties appear as attributes in the rendered HTML of a Lightning web component. Allowing properties to appear as attributes is especially important when creating accessible components, because screen readers and other assistive technologies use HTML attributes.

All HTML attributes are reactive by default. When an attribute's value changes in the component HTML, the component is re-rendered.

When you take control of an attribute by exposing it as a public property, the attribute no longer appears in the HTML output by default. To pass the value through to the rendered HTML as an attribute (to reflect the property), define a getter and setter for the property and call the `setAttribute()` method.

You can also perform operations in the setter. Use a private property to hold the computed value. Decorate the private property with `@track` to make the property reactive. If the property's value changes, the component rerenders.

This example exposes `title` as a public property. It converts the title to uppercase and uses the tracked property `privateTitle` to hold the computed value of the title. The setter calls `setAttribute()` to reflect the property's value to the HTML attribute.

```
/* mycomponent.js */
import { Element, api, track } from 'engine';

export default class MyComponent extends Element {
  @track privateTitle;
  @api
  get title() {
    return this.privateTitle;
  }
}
```

```

    set title(value) {
        this.privateTitle = value.toUpperCase();
        this.setAttribute('title', this.privateTitle);
    }
}

```

```

/* parent.html */
<template>
    <c-mycomponent title="Hover Over the Component to See Me"></c-mycomponent>
</template>

```

```

/* Generated HTML */
<c-mycomponent title="HOVER OVER THE COMPONENT TO SEE ME">
    <div>Reflecting Attributes Example</div>
</c-mycomponent>

```

To make sure that you understand how JavaScript properties reflect to HTML attributes, look at the same code without the call to `setAttribute()`. The generated HTML doesn't include the `title` attribute.

```

/* mycomponent.js */
import { Element, api, track } from 'engine';

export default class MyComponent extends Element {
    @track privateTitle;
    @api
    get title() {
        return this.privateTitle;
    }

    set title(value) {
        this.privateTitle = value.toUpperCase();
        // this.setAttribute('title', this.privateTitle);
    }
}

```

```

/* parent.html */
<template>
    <c-mycomponent title="Hover Over the Component to See Me"></c-mycomponent>
</template>

```

```

/* Generated HTML */
<c-mycomponent>
    <div>Reflecting Attributes Example</div>
</c-mycomponent>

```

Similarly, you can use `removeAttribute()` to hide HTML attributes from the rendered HTML.

SEE ALSO:

[Component Accessibility](#)

Compose Components


You can add components within the body of another component. This component composition enables you to build complex components from simpler building-block components.

In a large application, it's useful to compose the app with a set of smaller components to make the code more reusable and maintainable.

The `lightning` namespace contains many base components, such as `lightning-button`, that you can use to build your components.

Let's look at an app that composes components. The markup is contrived because we want to illustrate the concepts of *owner* and *container*. In a real app, the number of `c-todoitem` instances would be variable and populated dynamically in a `for:each` loop.

```
<!-- todoapp.html -->
<template>
  <c-todowrapper>
    <c-todoitem item-name="Milk"></c-todoitem>
    <c-todoitem item-name="Bread"></c-todoitem>
  </c-todowrapper>
</template>
```

 **Note:** The HTML spec mandates that tags for custom elements (components) aren't self-closing. Self-closing tags end with `/>`. Therefore, the `c-todoitem` component includes a separate closing `</c-todoitem>` tag.

Owner

The owner is the component that owns the template. In this example, the owner is the `c-todoapp` component. The owner controls all the composed components that it contains. The owner can:

- Set public properties on composed components
- Call methods on composed components
- Listen for any events fired by the composed components

Container

A container contains other components but itself is contained within the owner component. In this example, `c-todowrapper` is a container. A container is less powerful than the owner. A container can:

- Read, but not change, public properties in contained components
- Call methods on composed components
- Listen for some, but not necessarily all, events bubbled up by components that it contains.

Parent and child

When a component contains another component, which, in turn, can contain other components, we have a containment hierarchy. In the documentation, we sometimes talk about parent and child components. A parent component contains a child component. A parent component can be the owner or a container.

Owner Sets Public Properties

Let's look at how the owner, `c-todoapp`, sets public properties on the two instances of `c-todoitem`.

```
<c-todoitem item-name="Milk"></c-todoitem>
<c-todoitem item-name="Bread"></c-todoitem>
```

Pass data down the containment hierarchy from the owner to a composed component by setting an attribute in the composed component. In this example, the owner sets the `item-name` attribute of the `c-todoitem` components.

This example uses static values of `Milk` and `Bread`, but a real-world component would typically use a `for:each` iteration over a collection computed in the owner's JavaScript file, `todoapp.js`. For example:

```
<ul for:each={items} for:item="item">
  <li><c-todoitem item-name={item}></c-todoitem></li>
</ul>
```


The `items` collection is defined in `todoapp.js`.

```
@track items = ["Milk", "Bread"];
```

Now, let's look at how the `itemName` property is defined in the JavaScript class for `c-todoitem`.

```
/* todoitem.js */
import { Element, api } from "engine";
export default class TodoItem extends Element {
  @api itemName;
}
```

The `@api` decorator marks `itemName` as a public property. The owner can set the value for a child's public properties.

 **Note:** Property names in JavaScript are in camel-case while HTML attribute names are in kebab-case (dash-separated) to match HTML standards. In `todoapp.html`, the `item-name` attribute in markup maps to the `itemName` JavaScript property of `c-todoitem`.

Data Binding Between Components

When you add a component in markup, you can initialize public property values in the component based on property values of the owner component. The data binding for property values is one-way. If the property value changes in the owner component, the updated value propagates to the child component.

Use Slots as Placeholders

A slot in a component's HTML file is a placeholder for markup that a parent component passes into a component's body. A component can have zero or more slots.


SEE ALSO:

[Public Reactive Properties](#)

[Data Binding Between Components](#)

Data Binding Between Components

When you add a component in markup, you can initialize public property values in the component based on property values of the owner component. The data binding for property values is one-way. If the property value changes in the owner component, the updated value propagates to the child component.

 **Note:** The child component must treat any property values passed from the owner component as read-only. If the child component tries to change a value passed from an owner component, you see an error in the browser console.

To trigger a mutation for the property value provided by the owner component, the child component can send an event to the parent. If the parent owns the data, the parent can change the property value, which propagates down to the child component via the one-way data binding.

Example

The `c-todoapp` component sets the `itemName` public property in `c-todoitem` based on the `itemName` property in `c-todoapp`. This markup results in a data binding, also known as a value binding, between the two components. The `c-todoapp` component owns the data.

Here's the markup for `c-todoapp`.

```
<!-- todoapp.html -->
<template>
  <c-todowrapper>
    <c-todoitem item-name={itemName}></c-todoitem>
  </c-todowrapper>
  <p>item in todoapp: {itemName}</p>
  <p><button
    onclick={updateItemName}>Update item name in todoapp</button></p>
</template>
```

Here's the JavaScript file for `c-todoapp`.

```
/* c-todoapp.js */
import { Element, track } from 'engine';

export default class TodoApp extends Element {
  @track itemName = "Milk";

  updateItemName() {
    this.itemName = "updated item name in todoapp";
  }
}
```

Here's the markup for `c-todoitem`.

```
<!-- c-todoitem.html -->
<template>
  <p>item in todoitem: {itemName}</p>
  <p><button
    onclick={updateItemName}>Update item name in todoitem</button></p>
</template>
```

Here's the JavaScript file for `c-todoitem`.

```
/* c-todoitem.js */
import { Element, api } from "engine";
export default class TodoItem extends Element {
  @api itemName;


  // This code won't update itemName because:
  // 1) You can update public properties only at component construction time.
  // 2) Property values passed from owner components are read-only.
  updateItemName() {
    this.itemName = "updated item name in todoitem";
  }
}
```

Let's see how the one-way data binding works by clicking the **Update item name in todoapp** button in `c-todoapp`. The button triggers the `updateItemName()` method in `c-todoapp.js`, which updates `itemName` in `c-todoapp`.

In `c-todoapp`, we bound the `itemName` property (`item-name` in markup) in `c-todoitem` to the `itemName` property in `c-todoapp`.

```
<c-todoitem item-name={itemName}></c-todoitem>
```

Due to one-way data binding, when the `itemName` property is updated in the `c-todoapp` owner component, the `itemName` property in the `c-todoitem` child component is also updated.

 **Note:** Property names in JavaScript are in camel-case while HTML attribute names are in kebab-case (dash-separated) to match HTML standards. In `todoapp.html`, the `item-name` attribute in markup maps to the `itemName` JavaScript property of `c-todoitem`.

Now, let's see what happens when we update the `itemName` property in `c-todoitem` directly by clicking the **Update item name in todoitem** button.

Nothing happens in the UI! You see an error in the browser console saying that you can't change the value of `itemName`. You can't change `itemName` in `c-todoitem` because you can only set the value of a public property (`@api`) at component construction time.

Also, you can't work around this restriction by assigning data passed from an owner to a private reactive property (`@track`) and then modifying the private reactive property. For example, this code generates an error. A parent component, the data owner, sets a value for the `publicProp` property. Only the data owner can change the value passed to the `publicProp` property.

```
/* c-child.js */

export default class Child extends Element {
  @track _inner;
  @api set publicProp(value) {
    this._inner = value; // no error
    this._inner.foo = 'bar'; // error, cannot mutate because it's read-only
  }
}
```

One-way data binding makes code easier to maintain. Aura allows two-way data binding (bound expressions), which can lead to code complexity and unexpected side-effects when data changes value.

SEE ALSO:

[Compose Components](#)

[Data Binding Behavior Differences With Aura](#)

Use Slots as Placeholders

A slot in a component's HTML file is a placeholder for markup that a parent component passes into a component's body. A component can have zero or more slots.

Slots are part of the [Web Component specification](#).

A slot is defined in markup with the `<slot>` tag, which has an optional `name` attribute.

Beyond the Basics

In an Aura component, a facet is a similar concept to a slot.

Unnamed Slots

This example has an unnamed slot. The unnamed slot is a placeholder for any markup that a parent component passes into the body of `c-slotdemo`.

```
<!-- slotdemo.html -->
<template>
  <h1>Add content to slot</h1>
  <div>
    <slot></slot>
  </div>
</template>
```

Here's the markup for a parent component that uses `c-slotdemo`.

```
<!-- slotwrapper.html -->
<template>
  <c-slotdemo>
    <p>content from parent</p>
  </c-slotdemo>
</template>
```

When `c-slotdemo` is rendered, the unnamed slot is replaced with the markup passed into the body of `c-slotdemo`. Here's the rendered output of `c-slotwrapper`.

```
<h1>Add content to slot</h1>
<div>
  <p>content from parent</p>
</div>
```

If you have more than one unnamed slot, the markup passed into the body of the component is inserted into all the unnamed slots. This is an unusual UI pattern so a component usually has zero or one unnamed slots.

Named Slots

This example component has two named slots and one unnamed slot.

```
<!-- namedslots.html -->
<template>
  <p>First Name: <slot name="firstName">Default first name</slot></p>
  <p>Last Name: <slot name="lastName">Default last name</slot></p>
  <p>Description: <slot>Default description</slot></p>
</template>
```

Here's the markup for a parent component that uses `c-namedslots`.

```
<!-- slotswrapper.html -->
<template>
  <c-namedslots>
    <span slot="firstName">Willy</span>
    <span slot="lastName">Wonka</span>
    <span>Chocolatier</span>
  </c-namedslots>
</template>
```

`c-slotswrapper` drops:

- "Willy" into the `firstName` slot
- "Wonka" into the `lastName` slot
- "Chocolatier" into the unnamed slot

Here's the rendered output.

```
<p>First Name: <span slot="firstName">Willy</span></p>
<p>Last Name: <span slot="lastName">Wonka</span></p>
<p>Description: <span>Chocolatier</span></p>
```

SEE ALSO:

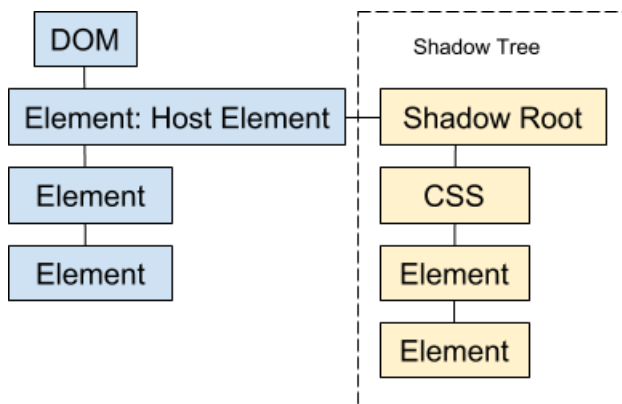
[Compose Components](#)

Shadow DOM

The DOM is the structure of the page containing a component. Shadow DOM is the web component standard that creates CSS and DOM encapsulation. Like other web components, a Lightning web component has its own shadow DOM to keep the component's styling and behavior consistent in any context.

A Lightning web component's shadow DOM works even on platforms that don't natively support shadow DOM, such as some web browsers. Lightning web components have shadow DOM enabled by default, so you don't have to do any work to implement it (no need to use the `attachShadow()` or `appendChild()` methods).

Think of the DOM as a tree. The shadow DOM is a tree that branches off the DOM. The shadow DOM subtree is called the shadow tree. The root of the shadow tree is called the shadow root. The element that the shadow root is attached to is called the host element.



The shadow tree is the structure that gives you encapsulation of styles and elements within the component. The container of the shadow DOM (such as the host document or parent component) can't introspect children elements within the shadow DOM. Be aware of the following when working with a component's shadow DOM.

Access Elements Owned by a Component

How do you access a component's shadow tree elements in code? Use `template` to access elements that the component owns.

```
this.template.querySelector();
this.template.querySelectorAll();
```

These methods allow you to look for the elements that your component rendered.

- The order of elements is not guaranteed.
- Elements not rendered to the DOM aren't returned in the `querySelector` result.

```
<!-- HTML file -->
<template>
  <div>First <slot name="task1">Task 1</slot></div>
  <div>Second <slot name="task2">Task 2</slot></div>
</template>

/* JavaScript file */
import { Element } from 'engine';

export default class Foo extends Element {
  renderedCallback() {
    this.template.querySelector('div'); // <div>First</div>
    this.template.querySelector('span'); // null
    this.template.querySelectorAll('div'); // [<div>First</div>, <div>Second</div>]
  }
}
```

CSS Encapsulation

Encapsulation is great for isolating a component's appearance and behavior. No matter the context, the component looks and behaves the same. For more information, see [Style Components with CSS](#).

Interact with Slots

The shadow DOM facilitates the use of slots. A slot identifies where to insert content you define some place else.

DOM elements that are passed to a component via slots aren't owned by the component and aren't in the component's shadow tree. To access these DOM elements passed in via slots, call `this.querySelector()` and `this.querySelectorAll()`. The component doesn't own these elements, so you don't use `template`.

The following example shows how to get the DOM elements passed to a child component from the child's context. Provide the selector name, such as an element, for `this.querySelector()` and `this.querySelectorAll()`.

```
<!-- child.html -->
<template>
  <div>First <slot name="task1">Task 1</slot></div>
  <div>Second <slot name="task2">Task 2</slot></div>
</template>

<!-- parent.html-->
<template>
  <c-child>
    <span slot="task1">push the green button.</span>
    <span slot="task2">push the red button.</span>
  </c-child>
</template>
```

```

    </c-child>
  </template>

```

```

// child.js
import { Element } from 'engine';

export default class Child extends Element {
  renderedCallback() {
    this.querySelector('span'); // <span>push the green button.</span>
    this.querySelectorAll('span'); // [<span>push the green button</span>, <span>push
the red button</span>]
  }
}

```

Using `element.querySelector` returns only elements that have been passed to your element via slots, exactly like using `this.querySelector` inside of your template. A call to `element.template.querySelector`, on the other hand, returns only elements that are defined inside of your template

```

<!-- mycomponent.html -->
<template>
  <div>
    <slot></slot>
  </div>
</template>

<my-component>
  <a href="#">Link</a>
</my-component>

myComponent.querySelector('div'); // null
myComponent.querySelector('a'); // <a>
myComponent.template.querySelector('div'); // <div>

```

For examples of working with slots and components, see [Use Slots as Placeholders](#).

Handle Events

Event targets don't propagate beyond the shadow root of the component instance. From outside the component, all event targets are the component itself. However, inside the shadow tree, you can handle events from specific targets in the tree. Depending on where you attach a listener for the event, and where the event happens, you might have different targets.

For more information on handling events, see [Handling an Event](#).

JavaScript Methods

Use a JavaScript method to communicate down the containment hierarchy. For example, a parent component calls a method on a child component that it contains. Methods are part of a component's API. Expose a method by adding the `@api` decorator to the method.

Communicate Between Components

To communicate up the containment hierarchy, fire an event in the child component with the `dispatchEvent` method, and handle it in the parent component.

Define a Method

This example exposes `isPlaying()`, `play()`, and `pause()` methods in a `c-videoplayer` component by adding the `@api` decorator to the methods. A parent component that contains `c-videoplayer` can call these methods. Here's the JavaScript file.

```
/* videoplayer.js */
import { Element, api } from 'engine';

export default class VideoPlayer extends Element {
  @api videoUrl;

  @api
  get isPlaying() {
    const player = this.template.querySelector('video');
    return player !== null && player.paused === false;
  }

  @api
  play() {
    const player = this.template.querySelector('video');
    // the player might not be in the DOM just yet
    if (player) {
      player.play();
    }
  }

  @api
  pause() {
    const player = this.template.querySelector('video');
    if (player) {
      // the player might not be in the DOM just yet
      player.pause();
    }
  }

  // private method for computed value
  get videoType() {
    return 'video/' + this.videoUrl.split('.').pop();
  }
}
```

`videoUrl` is a public reactive property. The `@api` decorator can be used to define a public reactive property, as well as a public JavaScript method, on a component. Public reactive properties are another part of the component's public API.



Note: Lightning Web Components use [CSS](#) to encapsulate components. To access the root of a component's shadow DOM tree, the code uses the `root` property.

Now, let's look at the HTML file where the video element is defined.

```
<!-- videoplayer.html -->
<template>
  <div class="fancy-border">
    <video autoplay>
      <source src={videoUrl} type={videoType} />
    </video>
  </div>
</template>
```

In a real-world component, `c-videoplayer` would typically have controls to play or pause the video itself. For this example to illustrate the design of a public API, the controls are in the parent component that calls the public methods.

Call a Method

The `c-methodcaller` component contains `c-videoplayer` and has buttons to call the `play()` and `pause()` methods in `c-videoplayer`. Here's the HTML.

```
<!-- methodcaller.html -->
<template>
  <div>
    <c-videoplayer video-url={video}></c-videoplayer>
    <button onclick={handlePlay}>Play</button>
    <button onclick={handlePause}>Pause</button>
  </div>
</template>
```

Clicking the buttons in `c-methodcaller` will play or pause the video in `c-videoplayer` after we wire up the `handlePlay` and `handlePause` methods in `c-methodcaller`.

Here's the JavaScript file for `c-methodcaller`.

```
/* methodcaller.js */
import { Element } from 'engine';

export default class MethodCaller extends Element {
  video = "https://www.w3schools.com/tags/movie.mp4";

  handlePlay() {
    this.template.querySelector('c-videoplayer').play();
  }

  handlePause() {
    this.template.querySelector('c-videoplayer').pause();
  }
}
```

The `handlePlay()` function in `c-methodcaller` calls the `play()` method in the `c-videoplayer` element. `this.root.querySelector('c-videoplayer')` returns the `c-videoplayer` element in `methodcaller.html`. The `this.template.querySelector()` call is useful to get access to a child component so that you can call a method on the component.

The `handlePause()` function in `c-methodcaller` calls the `pause()` method in the `c-videoplayer` element.

Return Values

Use the `return` statement to return a value from a JavaScript method. For example, see the `isPlaying()` method in `c-videoplayer`.

```
@api get isPlaying() {  
    const player = this.template.querySelector('video');  
    return player !== null && player.paused === false;  
}
```

Method Parameters

To pass data to a JavaScript method, define one or more parameters for the method. For example, you could define the `play()` method to take a `speed` parameter that controls the video playback speed.

```
@api play(speed) { ... }
```

SEE ALSO:

[Public Reactive Properties](#)

Share JavaScript Code

To share code between components, create an ES6 module and export the variables or functions that you want to expose.

An ES6 module is a file that explicitly exports functionality that other modules can use. Modules make it easier to structure your code without polluting the global scope.

Create Shared Code

To share code between components, create an ES6 module and use the standard `export` statement to export the functionality that your module exposes.

Your module can use the standard `import` statement to use other shared modules.

Use Shared Code

To use shared code in a JavaScript file, use the standard `import` statement. For example:

```
import { utils } from 'c-utils';
```

This line imports the `utils` module in the `c` namespace.



Note: You can't load JavaScript code from a static resource currently. For example, if you're developing an Aura component, you can reference JavaScript in a static resource using the `<ltng:require>` tag. This approach doesn't work for Lightning web components. We intend to provide a way to do this in the future.

Example

Let's create an ES6 module and see how it's used in a Lightning web component.

1. Start by creating a UseModule Salesforce DX project for the code.

```
cd path/to/your/sfdx/projects
sfdx force:project:create --projectname UseModule
cd UseModule/config
```

For more information on creating and configuring a project, see [Create an Empty Project for Lightning Web Components](#).

2. Edit the scratch-org definition file in `project-scratch-def.json` to look like this:

```
{
  "orgName": "Your Name Here",
  "edition": "Developer",
  "features": ["LightningWebComponents"],
  "orgPreferences" : {
    "enabled": ["S1DesktopEnabled"],
    "disabled": ["S1EncryptedStoragePref2"]
  }
}
```

3. Create a folder for the `utils` module.

```
cd path/to/your/sfdx/projects/UseModule
cd force-app/main/default/lightningcomponents
mkdir utils
cd utils
```

4. Create a `utils.js` file in the `utils` folder. This file is an ES6 module.

```
/* utils.js */
/**
 * Returns whether provided value is a function
 * @param {*} value - the value to be checked
 * @return {boolean} true if the value is a function, false otherwise
 */
export function isFunction(value) {
  return typeof value === 'function';
}
```

The `utils` module exports an `isFunction()` function that returns whether the provided parameter is a function.

5. Create a `utils.js-meta.xml` configuration file for the `utils` module.

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
  <apiVersion>43.0</apiVersion>
  <isExposed>false</isExposed>
</LightningComponentBundle>
```

6. Create a `c-libcaller` component that uses the module.

```
cd ..
mkdir libcaller
cd libcaller
```


7. Create a libcaller.js file.

```

/* libcaller.js */
import { Element, track } from 'engine';

// import the library
import { isFunction } from 'c-utils';

export default class LibCaller extends Element {
  @track result;

  checkType() {
    // call the imported library function
    this.result = isFunction(
      function() {
        console.log(" I am a function");
      }
    );
  }
}

```

The c-libcaller component imports the utils module and calls the isFunction function exported by the module. The argument passed into isFunction is a function so result is set to true.

8. Create libcaller.html.

```

<!-- libcaller.html -->
<template>
  <p>Is it a function?: {result}</p>

  <button onclick={checkType}>Check Type</button>
</template>

```

Clicking the button calls checkType(), which sets result to true.

9. Create a libcaller.js-meta.xml configuration file for the c-libcaller component.

```

<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
  <apiVersion>43.0</apiVersion>
  <isExposed>false</isExposed>
</LightningComponentBundle>

```

10. Create a folder for the Aura application that we use to run the component.

```

cd path/to/your/sfdx/projects/UseModule
cd force-app/main/default/aura
mkdir libApp
cd libApp

```

11. Create a libApp.app file.

```

<aura:application>
  <c:libcaller />
</aura:application>

```

The application includes the libcaller component.



Note: The syntax is `<c:libcaller />` with a colon separator as we're using the Lightning web component in an Aura application.

12. Authenticate with your dev hub, create a scratch org, and push your project to the scratch org.

```
sfdx force:auth:web:login -d -a LWC-Hub
sfdx force:org:create -s -f config/project-scratch-def.json -a "UseModule"
sfdx force:source:push
sfdx force:org:open
```

For more information on these commands, see [Create an Empty Project for Lightning Web Components](#).

13. Navigate to `https://myDomain.lightning.force.com/c/libApp.app`.
14. Click the button and confirm that the Lightning web component calls the function exported by the ES6 module.

SEE ALSO:

[Component JavaScript File](#)

Component Lifecycle

Lightning web components have a lifecycle managed by the framework. The framework creates components, inserts them into the DOM, renders them, and removes them from the DOM. It also monitors components for property changes.

[Component Lifecycle Hooks](#)

Lifecycle hooks are callback methods that let you run code at each stage of a component's lifecycle.

[Run Code When a Component Is Created](#)

The `constructor()` method fires when a component instance is created. Use the `constructor()` method to define properties and initialize reactive properties.

[Run Code When a Component Is Inserted or Removed from the DOM](#)

The `connectedCallback()` lifecycle hook fires when a component is inserted into the DOM. The `disconnectedCallback()` lifecycle hook fires when a component is removed from the DOM.

[Run Code When a Component Renders](#)

The `renderedCallback()` is unique to Lightning Web Components. Use it to perform logic after a component has finished the rendering phase.

[Handle Component Errors](#)

The `errorCallback()` is unique to Lightning Web Components. Implement it to create an error boundary component that captures errors in all the descendent components in its tree. You can code the error boundary component to log stack information and render an alternative view to tell users what happened and what to do next.

Component Lifecycle Hooks

Lifecycle hooks are callback methods that let you run code at each stage of a component's lifecycle.

constructor()

Called when the component is created. This hook flows from parent to child. You can't access child elements in the component body because they don't exist yet. Properties are not passed yet, either. Properties are assigned to the component after construction and before the `connectedCallback()` hook. You can access the host element with `this.root`.

connectedCallback()

Called when the element is inserted into a document. This hook flows from parent to child. You can't access child elements in the component body because they don't exist yet. You can access the host element with `this.root`.

disconnectedCallback()

Called when the element is removed from a document. This hook flows from parent to child.

render()

For complex tasks like conditionally rendering a template or importing a custom one, use `render()` to override standard rendering functionality. This function gets invoked after `connectedCallback()` and must return a valid HTML template.

renderedCallback()

Called after every render of the component. This lifecycle hook is specific to Lightning Web Components, it isn't from the HTML custom elements specification. This hook flows from child to parent.

If you use `renderedCallback()` to perform a one-time operation, you must track it manually (using an `initialRender` private property, for example). If you perform changes to reactive attributes, guard them or they can trigger wasteful rerenders or an infinite rendering loop.

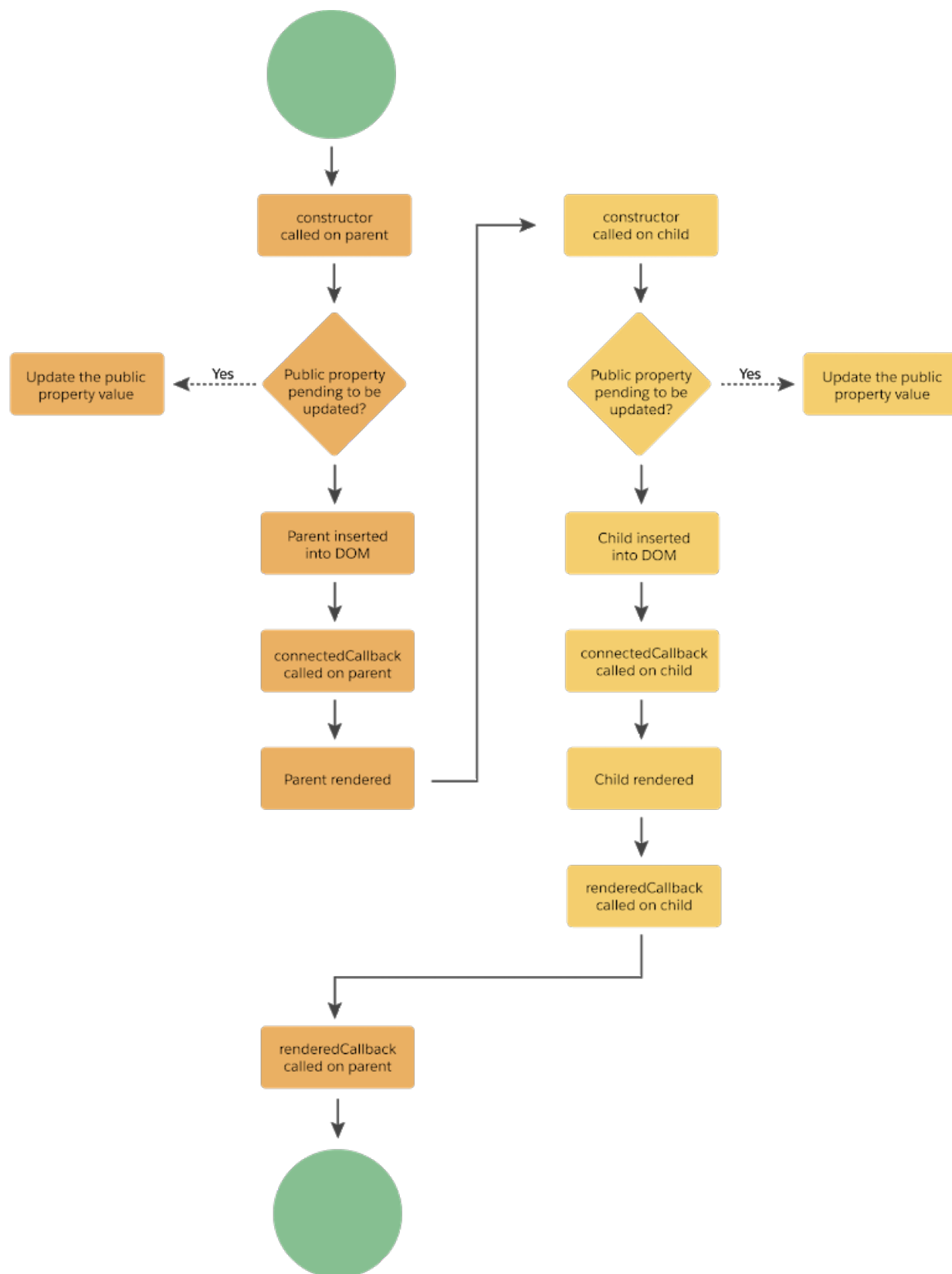
errorCallback(error, stack)

Called when a descendant component throws an error in one of its lifecycle hooks. The `error` argument is a JavaScript native error object, and the `stack` argument is a string. This lifecycle hook is specific to Lightning Web Components, it isn't from the HTML custom elements specification.

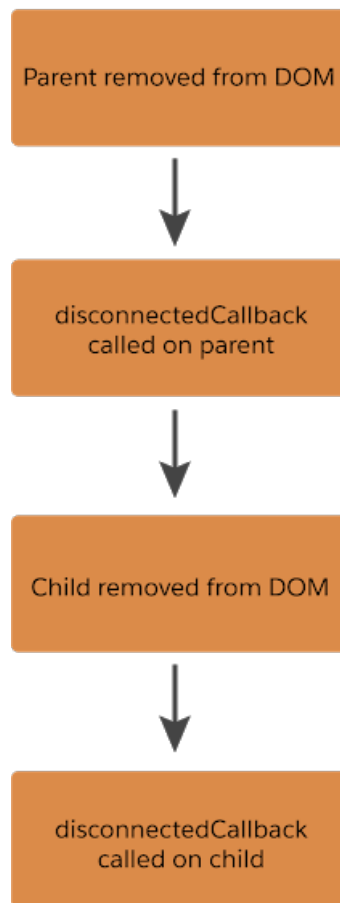
Implement this hook to create an error boundary component that captures errors in all the descendent components in its tree. The error boundary component can log stack information and render an alternative view to tell users what happened and what to do next. The method works like a JavaScript `catch{ }` block for components that throw errors in their lifecycle hooks. It's important to note that an error boundary component catches errors only from its children, and not from itself.

Lifecycle Flow

This diagram shows the flow of the component lifecycle from creation through render.



This diagram shows what happens when a component instance is removed from the DOM.



Run Code When a Component Is Created

The `constructor()` method fires when a component instance is created. Use the `constructor()` method to define properties and initialize reactive properties.

The constructor flows from parent to child.

This code is from the `todo.js` file in the [sfdx-lwc-samples repo](#).

```
/* todo.js */

export default class Todo extends Element {
  @track todos;

  @track filter;

  constructor() {
    super();
    this.todos = store.load();
    this.filter = getCurrentFilter();
    window.addEventListener('hashchange', () => (this.filter = getCurrentFilter()));
  }
}
```

```

    }

    // Code removed to keep example short.
    // Complete code is in the sfdx-lwc-samples repo.
}

```

These requirements from the [HTML: Custom elements spec](#) apply to the Lightning web components `constructor()`.

- The first statement must be `super()` with no parameters. This call establishes the correct prototype chain and value for `this`. Always call `super()` before touching `this`.
- Don't use a `return` statement inside the constructor body, unless it is a simple early-return (`return` or `return this`).
- Don't use the `document.write()` or `document.open()` methods.
- Don't inspect the element's attributes and children, because they don't exist yet.
- Don't inspect the element's public properties, because they're set after the component is created.

Do Not Add Attributes to Host Element During Construction

You can add attributes to the host element during any stage of the component lifecycle other than construction.

This example is legal, because it adds an attribute to the host element in the `connectedCallback()`.

```

import { Element } from 'engine';

export default class New extends Element {
  connectedCallback() {
    this.classList.add('new-class');
  }
}

```

This example is illegal, because it adds an attribute to the host element in the `constructor()`.

```

import { Element } from 'engine';

export default class Deprecated extends Element {
  constructor() {
    super();
    this.classList.add('new-class');
  }
}

```

Run Code When a Component Is Inserted or Removed from the DOM

The `connectedCallback()` lifecycle hook fires when a component is inserted into the DOM. The `disconnectedCallback()` lifecycle hook fires when a component is removed from the DOM.

Use these hooks to do all the bookkeeping related to the DOM.

Both hooks flow from parent to child. You can't access child elements from the callbacks because they don't exist yet. You can access the host element with `this.template`.

The `connectedCallback()` hook can fire more than once. For example, if you remove an element and then insert it into another position, such as when you reorder a list, the hook fires several times. If you want code to run one time, write code to prevent it from running twice.

One common use for these hooks is to add and remove event listeners.

```
import { Element } from 'engine';
export default class Foo extends Element {
  connectedCallback() {
    this.addEventListener('mouseenter', this.lightUp);
    this.addEventListener('mouseleave', this.lightDown);
  }
  disconnectedCallback() {
    this.removeEventListener('mouseenter', this.lightUp);
    this.removeEventListener('mouseleave', this.lightDown);
  }
  lightUp = () => {
    this.classList.add('make-container-opaque');
  }
  lightDown = () => {
    this.classList.remove('make-container-opaque');
  }
}
```



Tip: Removing listeners is a good practice, but not strictly necessary since Lightning Web Components cleans remaining listeners when destroying a host element.

Run Code When a Component Renders

The `renderedCallback()` is unique to Lightning Web Components. Use it to perform logic after a component has finished the rendering phase.

This hook flows from child to parent.

Due to mutations, the component is usually rendered many times during its lifespan in an application. If you use this hook to perform a one-time operation, use a private property like `hasRendered` to track it manually.

This code is `todoitem.js` from the [sfdx-lwc-samples](#) repo. In this code, if the private `editing` property is `true`, focus is assigned to the input field with `class="edit"`.

```
/* todoitem.js */

import { Element, api, track } from 'engine';
import { ENTER_KEY, ESCAPE_KEY } from 'c-todo_utils';

export default class TodoItem extends Element {
  @track editing = false;

  @track _todo;

  @api
  get todo() {
    return this._todo;
  }

  @api
  set todo(newValue) {
    this.classList[newValue.completed ? 'add' : 'remove']('completed');
    this._todo = newValue;
  }
}
```

```
}

fireUpdate() {
  const title = this.root.querySelector('input.edit').value.trim();
  const completed = this.root.querySelector('input.toggle').checked;
  const detail = { title, completed };
  const event = new CustomEvent('update', { detail });
  this.dispatchEvent(event);
}

fireRemove() {
  const event = new CustomEvent('remove');
  this.dispatchEvent(event);
}

handleCompletedInput() {
  this.fireUpdate();
}

handleRemoveInput() {
  this.fireRemove();
}

handleEditModeInput() {
  this.editing = true;
  // view vs edit elements are toggled via css
  this.classList.add('editing');
}

handleBlur() {
  this.editing = false;
  // view vs edit elements are toggled via css
  this.classList.remove('editing');
}

handleTitleInput(evt) {
  const title = evt.target.value.trim();
  if (!title) {
    // remove todo if title is cleared
    this.fireRemove();
    return;
  }
  this.fireUpdate();
}

handleKeyDown(evt) {
  const { keyCode } = evt;
  if (keyCode === ENTER_KEY || keyCode === ESCAPE_KEY) {
    const el = this.root.querySelector('input.edit');
    // [esc] cancels the edit
    if (keyCode === ESCAPE_KEY) {
      el.value = this.todo.title;
    }
    // [return] saves the edit
  }
}
```



```

        el.blur();
      }
    }

    renderedCallback() {
      if (this.editing) {
        this.root.querySelector('input.edit').focus();
      }
    }
  }
}

```

Here's the `todo_item` HTML template.

```

<!-- todo_item.html -->

<template>
  <div class="view">
    <input class="toggle" type="checkbox" checked={todo.completed}
    onchange={handleCompletedInput}>
    <label onclick={handleEditModeInput}>{todo.title}</label>
    <button class="destroy" onclick={handleRemoveInput}></button>
  </div>
  <input class="edit" type="text" value={todo.title}
  onblur={handleBlur}
  onchange={handleTitleInput}
  onkeydown={handleKeyDown} />
</template>

```

Handle Component Errors

The `errorCallback()` is unique to Lightning Web Components. Implement it to create an error boundary component that captures errors in all the descendent components in its tree. You can code the error boundary component to log stack information and render an alternative view to tell users what happened and what to do next.

You can create an error boundary component and reuse it throughout an app. It's up to you where to define those error boundaries. You can wrap the entire app, or every individual component. Most likely, your architecture will fall somewhere in between. A good rule of thumb is to think about where you'd like to tell users that something went wrong.

This example implements the `errorCallback()` method.

```

<!-- boundary.html -->

<template>
  <template if:true={this.error}>
    <error-view error={this.error} info={this.stack}></error-view>
  </template>
  <template if:false={this.error}>
    <healthy-view></healthy-view>
  </template>
</template>

```

```

// boundary.js

import { Element, track } from 'engine';
export default class Boundary extends Element {

```

```

    @track error;
    @track stack;
    errorCallback(error, stack) {
        this.error = error;
    }
}

```

You don't have to use `if: [true | false]` in a template. For example, let's say you define a single component template. If this component throws an error, the framework calls `errorCallback` and unmounts the component during re-render.

```

<!-- boundary.html -->

<template>
  <my-one-and-only-view></my-one-and-only-view>
</template>

```

Labels and Static Resources

Lightning components can access global Salesforce values, such as labels and resources.

This functionality is equivalent to Aura's [global value providers](#).

Resources

To reference images, style sheets, archives, and JavaScript code that you've uploaded in static resources, use `@salesforce/resource-url` in an `import` statement.

Labels

Custom labels are text values stored in Salesforce that can be translated into any language that Salesforce supports. Use custom labels to create multilingual applications that present information (for example, help text or error messages) in a user's native language.

Get Current User

To access the current user Id, import `@salesforce/user/Id` in a component's JavaScript class. Note that `@salesforce/user/` cannot be imported by itself; you must indicate the property that you want to import. For the User standard object, only the Id field is available.

Resources

To reference images, style sheets, archives, and JavaScript code that you've uploaded in static resources, use `@salesforce/resource-url` in an `import` statement.

```
import myResource from '@salesforce/resource-url/resource-reference';
```

```
import myResource from '@salesforce/resource-url/namespace__managed-resource-reference'
```

The value of `resource-reference` is the name of the static resource.

A static resource name can contain only underscores and alphanumeric characters, and must be unique in your org. It must begin with a letter, not include spaces, not end with an underscore, and not contain two consecutive underscores.

The value of `managed-resource-reference` is the name of the static resource in a managed package.

Let's look at a sample from the `gallery_static_resources` component in the [sfdx-lwc-samples repo](#).

To reference a resource in a template, use `{property}` syntax, which is the same syntax you use to reference any JavaScript property.

```

<!-- gallery_static_resources.html -->

<template>
  <c-card header="Static Resources">
    <p class="slds-hyphenate slds-m-vertical_large">
      This sample shows how to reference static resources and custom labels. See the
      source code at <a
href={gitSource}><code>force-app/main/default/lightningcomponents/gallery_static_resources</code></a>.

    </p>
    <p class="slds-m-vertical_large">
      The Salesforce logo is a static resource. The alt text is defined in a custom
      label.<br>
      <img src={salesforceLogoUrl} alt={label.salesforceLogoDescription} width=100>

    </p>
    <p class="slds-m-vertical_large">
      This image is nested in a static resource ZIP archive. The alt text is defined
      in a custom label.<br>
      <img src={avatarUrl} alt={label.avatarDescription} width=70>
    </p>
  </c-card>
</template>

```

The JavaScript code imports a logo resource.

```

// gallery_static_resources.js

import { Element } from 'engine';
import { getGitPath } from 'c-utils';

// Import the URL for an image and zip stored as static resources
import salesforceLogo from '@salesforce/resource-url/salesforce_logo';
import avatarArchive from '@salesforce/resource-url/avatar_archive';

// Import custom labels
import greeting from '@salesforce/label/c.greeting';
import salesforceLogoDescription from '@salesforce/label/c.salesforce_logo_description';
import avatarDescription from '@salesforce/label/c.avatar_description';

/**
 * Sample demonstrating referencing static resources and custom labels.
 */
export default class GalleryStaticResources extends Element {
  // expose the static resource URL for use in the template
  salesforceLogoUrl = salesforceLogo;
  // concatenate the path for an item in an archive
  avatarUrl = avatarArchive + '/assets/images/avatar1.png';

  // expose the labels for use in the template
  label = {

```

```

        greeting,
        salesforceLogoDescription,
        avatarDescription,
    };

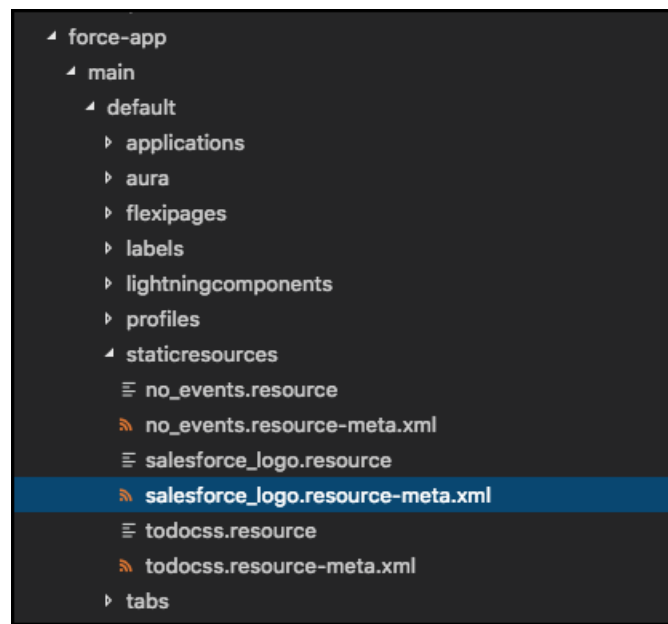
    get gitSource() {
        return getGitPath('/lightningcomponents/gallery_static_resources');
    }
}

```

A static resource can be an archive file with a nested directory structure. To reference an item in an archive, concatenate a string to create the path to the item, as the example does to build `avatarUrl`.

Place the resources in a `staticresources` directory in your app. You can't create subdirectories of `staticresources`.

Create a `.resource-meta` metadata file for each resource. The metadata file defines the [content type](#) of the resource.



SEE ALSO:

https://help.salesforce.com/articleView?id=pages_static_resources.htm

Labels

Custom labels are text values stored in Salesforce that can be translated into any language that Salesforce supports. Use custom labels to create multilingual applications that present information (for example, help text or error messages) in a user's native language.

To create custom labels, from Setup, enter *Custom Labels* in the Quick Find box, then select **Custom Labels**.

To import a label in a Lightning Web Component JavaScript file, use `@salesforce/label` in an `import` statement.

```
import labelName from '@salesforce/label/label-reference';
```

The `label-reference` is the label name and must include a namespace in the format `namespace.labelName`.

This sample code imports two labels. Play with the code in the [sfdx-lwc-samples repo](#).

```
/* other.js */

import { Element } from 'engine';

// Import the URL for the static resource named 'salesforce_logo'
import salesforceLogo from '@salesforce/resource-url/salesforce_logo';

// Import custom labels
import greeting from '@salesforce/label/c.greeting';
import salesforceLogoDescription from '@salesforce/label/c.salesforce_logo_description';

export default class Other extends Element {
  // expose the static resource URL for use in the template
  logoUrl = salesforceLogo;

  // expose the labels for use in the template
  label = {
    greeting,
    salesforceLogoDescription,
  };
}
```

To use the labels in the template, use the same `{property}` syntax that you use to reference any JavaScript property.

```
<!-- other.html -->

<template>
  <c-page_header header="Using static resources and custom labels" description="This
sample shows how to reference external items like static resources and custom
labels"></c-page_header>

  <c-card>
    <img src={logoUrl} alt={label.salesforceLogoDescription} width=100><br>
    <br>
    {label.greeting}
  </c-card>
</template>
```

SEE ALSO:

https://help.salesforce.com/articleView?id=cl_about.htm

Get Current User


To access the current user Id, import `@salesforce/user/Id` in a component's JavaScript class. Note that `@salesforce/user/` cannot be imported by itself; you must indicate the property that you want to import. For the User standard object, only the Id field is available.

```
import { Id } from '@salesforce/user/';
```

This sample code imports the current user ID by using a scoped module import. It passes the ID to the `getRecord` wire adapter and the wire service provisions the User record.

```
/* other.js */

import UserId from '@salesforce/user/Id';
import FirstNameField from '@salesforce/schema/User.FirstName';
import LastNameField from '@salesforce/schema/User.LastName';
import { getRecord } from 'lightning-ui-api-record';
import { Element, wire } from 'lwc';
export default class Example extends Element {
  @wire(getRecord, {recordId: UserId, fields: [FirstNameField, LastNameField]})
  wiredUser;
}
```

 **Tip:** If you're familiar with Visualforce, `@salesforce/user/Id` is the equivalent of `$User.Id`.

Component Accessibility

Screen readers and other accessibility software interpret UI elements for users by reading their HTML attributes, like `title` or `role`. Create components that help accessibility software do its job.

One critical piece of accessibility is the use of the `title` attribute. Screen readers read `title` attribute values to a user. When you consume a Lightning web component with a `title` attribute, always specify a value. For example, the `lightning-button` component has a `title` attribute.

```
/* parent.html */
<template>
  <lightning-button title="Log In" label="Log In" onclick={login}></lightning-button>
</template>
```

That template creates HTML output like the following for the screen reader to read out "Log In" to the user.

```
<!-- Generated HTML -->
<lightning-button>
  <button title="Log In">Log In</button>
</lightning-button>
```

When you're creating a Lightning web component, use `@api` to expose a public `title` attribute if you want a screen reader to read a value aloud to the user.

When you take control of an attribute by exposing it as a public property, the attribute no longer appears in the HTML output by default. To pass the value through to the rendered HTML as an attribute (to reflect the property), define a getter and setter for the property and call the `setAttribute()` method.

You can also perform operations in the setter. Use a private property to hold the computed value. Decorate the private property with `@track` to make the property reactive. If the property's value changes, the component rerenders.

```
/* mycomponent.js */
import { Element, api, track } from 'lwc';

export default class MyComponent extends Element {
  @track privateTitle;
  @api
  get title() {
```

```

        return this.privateTitle;
    }

    set title(value) {
        this.privateTitle = value.toUpperCase();
        this.setAttribute('title', this.privateTitle);
    }
}

```

```

/* parent.html */
<template>
    <c-mycomponent title="Hover Over the Component to See Me"></c-mycomponent>
</template>

```

```

/* Generated HTML */
<c-mycomponent title="HOVER OVER THE COMPONENT TO SEE ME">
    <div>Reflecting Attributes Example</div>
</c-mycomponent>

```

For more information on using `setAttribute()`, see [Reflect JavaScript Properties to HTML Attributes](#).

ARIA Attributes

To provide more advanced accessibility, like have a screen reader read out a button's current state, use [ARIA](#) attributes. These attributes give more detailed information to the screen readers that support the ARIA standard.



Note: ARIA attributes use camel-case in accessor functions. For example, `aria-label` becomes `ariaLabel`. The complete mapping list is in the Github repo at [lwc/packages/lwc-template-compiler/src/parser/constants.ts](https://github.com/salesforce/lwc/blob/master/packages/lwc-template-compiler/src/parser/constants.ts)

For example, the `aria-pressed` attribute tells screen readers to say when a button is pressed. When using a `lightning-button` component, you write:

```

/* parent.html */
<template>
    <lightning-button title="Log In" label="Log In" onclick={login} aria-label="Log In"
    aria-pressed="true"></lightning-button>
</template>

```

The component defines the ARIA attributes as public properties, and uses private reactive properties to get and set the public properties.

```

/* lightning-button.html */
<template>
    <button title="Log In" label="Log In" onclick={login} aria-label={innerLabel}
    aria-pressed={pressed}></button>
</template>

```

The component Javascript uses the camel-case attribute mappings to get and set the values in `lightning-button.js`.

```

/* lightning-button.js */
import { Element, api, track } from 'engine';
export default class LightningButton extends Element {
    @track innerLabel;

    @api
    set ariaLabel(newValue) {

```

```

        this.innerLabel = newValue;
    }

    @api
    get ariaLabel() {
        return this.innerLabel;
    }

    @track pressed;

    @api
    set ariaPressed(newValue) {
        this.pressed = newValue;
    }

    @api
    get ariaPressed() {
        return this.pressed;
    }
}

```

So the generated HTML is:

```

<lightning-button>
  <button title="Log In" label="Log In" onclick={login} aria-label="Log In"
aria-pressed="true"></button>
</lightning-button>

```

A screen reader that supports ARIA reads the label and indicates that the button is pressed.


Default ARIA Values

A component author may want to define default ARIA attributes on a custom component, and still allow component consumers to specify attribute values. In this case, a component author defines default ARIA values on the component's shadow root.

```

/* lightning-button.js sets "Log In" as the default label */
import { Element } from 'engine';
export default class LightningButton extends Element {
    connectedCallback() {
        this.root.ariaLabel = 'Log In';
    }
}

```

 **Note:** Define attributes in `connectedCallback()`. Don't define attributes in `constructor()`.

When you use the component and supply an `aria-label` value, the supplied value appears.

```

/* parent.html */
<template>
  <lightning-button title="Log In" label="Submit" onclick={login} aria-label="Submit"
aria-pressed="true"></lightning-button>
</template>

```


The generated HTML is:

```
<lightning-button>
  <button title="Log In" label="Submit" onclick={login} aria-label="Submit"
  aria-pressed="true"></button>
</lightning-button>
```

And, when you don't supply an `aria-label` value, the default value appears.

```
/* parent.html */
<template>
  <lightning-button title="Log In" label="Log In" onclick={login}></lightning-button>
</template>
```

The generated HTML is:

```
<lightning-button>
  <button title="Log In" label="Log In" onclick={login} aria-label="Log In"></button>
</lightning-button>
```

Static Values

What if you create a custom component and don't want the value of an attribute to change? A good example is the `role` attribute. You don't want a component consumer to change `button` to `tab`. A button is a button.

You always want the generated HTML to have the `role` be `button`, like in this example.

```
<lightning-button>
  <div title="Log In" label="Log In" onclick={login} role="button"></div>
</lightning-button>
```

To prevent a consumer from changing an attribute's value, simply return a string. This example always returns `"button"` for the `role` value.

```
/* lightning-button.js */
import { Element, api } from 'engine';
export default class LightningButton extends Element {
  @api
  set role(value) {}

  @api
  get role() { return "button"; }
}
```

SEE ALSO:

[Component Lifecycle Hooks](#)

CHAPTER 3 Communicate with Events

In this chapter ...

- [Simple Events](#)
- [Events with Data](#)
- [Extended Event Types](#)
- [Creating an Event](#)
- [Dispatching an Event](#)
- [Handling an Event](#)
- [Understanding Event Propagation](#)
- [Sending Events to an Enclosing Aura Component](#)

Use events to communicate up the containment hierarchy. For example, a component can fire an event up the DOM tree to communicate with its parent elements. Public events and event handlers should be considered a part of a component's API.

Events in Lightning web components are built upon the Document Object Model (DOM) standard, a collection of APIs and objects available in every browser. Use the `Event` object or one of its more specific interfaces to create events. Lightning web components implement the `EventTarget` interface, which allows them to fire, listen for, and handle fired events.

The [DOM event system](#) is, at its core, simply a programming design pattern. The pattern starts with an agreement over a kind of event, along with the following elements.

- The name of the event, a string that identifies the event, and is used by event listeners to identify events of interest
- The data structure used to represent key properties of the event
- The JavaScript object that emits the event

This design pattern is enriched by interfaces that support different kinds of events, from simple and general to specific and structured. And it's supported by an event propagation system that's simple but very flexible.

Using events is the recommended way of communication in Lightning web components. Using the event model encourages encapsulation, it's a standard that is well understood by web app developers, and (used correctly) it's lightweight and efficient.

Simple Events

To create and fire an event, such as a notification, in your component use the `Event` constructor to create the event, and the `EventTarget.dispatchEvent()` method to send it.

Here's an example of creating and firing an event from a custom component.

```
import { Element } from 'engine';

export default class SimpleNotification extends Element {
  handleNotify() {
    // Create a simple notification event
    const myEvent = new Event('notification', {
      bubbles: true,
      cancelable: true,
      composed: true
    });

    // Fire the event
    this.dispatchEvent(myEvent);
  }
}
```

This notification is a simple “something happened” kind of event. It carries no data payload. To add data to the `myEvent` object, use a `CustomEvent` instead.

See [Creating an Event](#) for more details on creating different kinds of events. You can find the complete list of initialization properties for an event (`bubbles`, `cancelable`, and so on) in the [Event interfaces doc on MDN](#), or directly in the [DOM specification](#).

Events with Data

To fire an event with a data payload, use the `CustomEvent` constructor to create the event, and the `EventTarget.dispatchEvent()` method to send it.

`CustomEvent` extends the `Event` interface, and allows setting a `detail` property as part of the initialization. This event is useful for passing data up to the receiving component.

Here's an example of creating a `CustomEvent` with a simple data payload.

```
import { Element } from 'engine';

export default class NotificationWithData extends Element {

  handleNotifyWithPriority() {
    // Create a notification event with data in detail property
    const myEventWithPriority = new CustomEvent('notification', {
      bubbles: true,
      cancelable: true,
      composed: true,
      detail: { priority: 'URGENT' }
    });

    // Fire the event
    this.dispatchEvent(myEventWithPriority);
  }
}
```

```

    }
  }
}

```

`CustomEvent` allows receiving components to access the additional data from within the event listener's handler function. For example:

```

function eventHandler(e) {
  console.log('The notification priority is ', e.detail.priority);
}

```

The `CustomEvent` interface imposes no type requirements or structure on the `detail` property. You can put anything you want into it, from a simple type like a number or string, to complex structured objects. The only "requirement" is that sender and receiver agree on what they expect to add to or find in the `detail` property.



Warning: If you include complex data in `detail`, be aware that structured objects are passed by reference in JavaScript. The event receiver can potentially modify (mutate) objects passed by reference, and allowing it is an anti-pattern. If the object references private state of the event sender then the sender has leaked private mutable state, which is a bigger anti-pattern. It's a best practice to copy such data to a new object before adding to an event property. This ensures that you're only sending the data you want, and that the receiver can't mutate your data unexpectedly.

In most situations using the `CustomEvent` interface is the way to go. An event using this interface requires no setup or boilerplate, it's simple and standard, and it allows you to pass any kind of data via the `detail` property, which makes it very flexible.

Extended Event Types

If your event has specific semantics or constraints, or well-defined attributes, it can make sense to create your own specialized event type. Extend the `Event` class to create your own type of event.

For example, the browser [extends the `Event` interface for a number of specialized events](#), mostly dealing with user interface events. For example [MouseEvent](#) or [TouchEvent](#).

Here's an example of how to build your own type of event for navigation within Lightning Experience. This `NavigateToPageEvent` has specific semantics, and requires a clear, constrained set of attributes.

```

import { Element } from 'engine';
export const MySpecialEventName = 'mynamespace_myspecialevent';

export default class MySpecialEvent extends Event {
  constructor({ myType, myData }) {
    super(MySpecialEventName, {
      composed: true,
      cancelable: false,
      bubbles: true,
    });

    this.myType = myType || 'special';
    this.myData = myData || {};
  }
}

```

Extending `Event` can be very powerful, since it allows you to control the shape and the semantics of the properties you define. It also requires more setup than simply using `Event` or `CustomEvent`.



Tip: As a rule of thumb, unless your event has a specific set of attributes and semantics, we recommend using `CustomEvent` since it's simple and practical.

Creating an Event

To create an event, regardless of the type, use the event constructor, providing it with an event name, and a set of initial values that define its behavior and shape.

The following sections describes each of these, as well as provides some good practices and common patterns.

Event Constructors

The following example illustrates using the relevant constructor to create a `CustomEvent` as described in [Events with Data](#), and a `MySpecialEvent` as described in [Extended Event Types](#).

```
const myEventWithPriority = new CustomEvent('notification', { detail: { priority: 'URGENT' } });
const specialEvent = new MySpecialEvent({ myType: 'partyTime' });
```

Event Initialization Properties

Here are the most common properties for an event and the recommended values and use cases. Event initialization properties inherit from the [DOM Event API](#), which you should review for extended details of initialization properties.

bubbles

A Boolean indicating whether the event bubbles up through the DOM or not. Defaults to `false`.

When an event “bubbles” up it means that container elements of the element that dispatched the event can observe it. Normally you set this attribute to `true`, unless you are expecting to handle the event in the same element on which it’s dispatched.

If you set this attribute to `true`, you must also set the `composed` attribute to `true`.

composed

A Boolean value indicating whether or not the event can escape the shadow root that the component instance belongs to. Defaults to `false`.

detail

A property specific to the `CustomEvent` interface of type any (that is, any data type). Optional and defaults to `null`.

This property is an event-dependent value associated with the event. Use this property to attach data that’s relevant for the consumer of the event.

To create an event to send up and out from your component, set `bubbles` and `composed` to `true`. To create an event that your component sends only to itself, set `bubbles` and `composed` to `false`.




Note: Any event that crosses the element boundary (`bubbles` and `composed` set to `true`) is a part of your component’s public API. There’s no explicit decoration, like `@api` for a component method. Be careful, and double check which events you want to expose **and maintain forever**.

Event Names

You can use any arbitrary string as your event name. However, we recommend that you conform with the DOM event standard: No uppercase, no spaces, use an underscore when needed for separation.



Tip: If you’re firing an event beyond the element boundary (`bubbles` and `composed` set to `true`), we recommend that you prefix it with your namespace to avoid unexpected listeners. This is to prevent name collisions, which won’t generate a warning or error, just unexpected behavior.

 **Warning:** Never prefix your event name with the string “on”. That results in extremely confusing markup within consumers of the event. For example, `<x-foo ononfoo={handleFoo}>` (notice the doubled word “onon”). Another developer will see “ononfoo”, think it’s a mistake, and “fix” it for you. That’ll be fun to debug.

SEE ALSO:

[Understanding Event Propagation](#)

[Shadow DOM](#)

Dispatching an Event

To dispatch an event, call the `dispatchEvent` method on the `Element` instance (that is, your component), and pass in the event instance.

Dispatching an event is also called firing an event, or sending an event. The terms are in most cases interchangeable. Here’s an example of dispatching a simple event, with the relevant line emphasized.

```
import { Element } from 'engine';

export default class NotificationWithData extends Element {

  handleNotifyWithPriority() {
    // Create a notification event with data in detail property
    const myEventWithPriority = new CustomEvent('notification', {
      bubbles: true,
      cancelable: true,
      composed: true,
      detail: { priority: 'PRIORITY_URGENT' }
    });

    // Fire the event
    this.dispatchEvent(myEventWithPriority);
  }
}
```

Remember that by default an event only bubbles up to the custom element boundary (the component’s root). If you want the event to bubble further, set the event’s `bubbles` and `composed` properties to `true` when you create it.

Handling an Event

There are two ways to listen for a particular event: declaratively from the HTML template, or programmatically using an imperative API.

Attach an Event Listener Declaratively

The listener is declared in markup in the template of the owner component, `c-handler`.

```
<!-- handler.html -->
<template>
  <p>Outermost Handler component</p>
  <c-wrapper>
    <c-dispatcher onnotification={handleNotification}></c-dispatcher>
  </c-wrapper>
</template>
```

```

    </c-wrapper>
</template>

```

The event handler function, `handleNotification`, is defined in the JavaScript file of `c-handler`.

Attach an Event Listener Programmatically

Both the listener and the handler function are defined in the JavaScript file of the `c-wrapper` component.

```

export default class Wrapper extends Element {
  constructor() {
    super();
    this.template.addEventListener('notification', this.handleNotification.bind(this));
  }

  handleNotification() {
    // implement handler logic here
  }
}

```

Which Technique Should You Use?

A declarative event listener does not behave the same as a programmatic event listener. The declarative method always triggers, regardless of the configuration of the `Event` in `c-dispatcher`. The imperative handler added in code only reaches that level if the event is set to bubble beyond the component that fires it by setting `{bubbles: true, composed: true}` when creating the event.

Event Target

The `Event.target` property is a reference to the object that dispatched an event. `Event.target` is part of the DOM API for events.

When an event propagates from a component that dispatches the event up to a parent component, the event passes the shadow DOM boundary. The event is retargeted to avoid exposing internal details of the component that dispatched the event. The target value of the event is changed to the custom element (component) that dispatched the event. The event doesn't point at the tag within the custom element that dispatched the event.

Let's look at the example of a `c-dispatcher` component that dispatches a notification event. The event is handled by the `c-dispatcher`, `c-wrapper`, and `c-handler` components. You can see how the event target changes when the event propagates through the shadow DOM boundary.

```

<!-- handler.html -->
<template>
  <p>Outermost Handler component</p>
  <c-wrapper>
    <c-dispatcher onnotification={handleNotification}></c-dispatcher>
  </c-wrapper>
</template>

```

Here's the template for `c-dispatcher`.

```

<!--dispatcher.html-->
<template>
  <div>Composed component dispatches event</div>

```

```
<p><button onclick={dispatchNotificationEvent}>Dispatch event</button></p>
</template>
```

Here's the JavaScript file for `c-dispatcher`.

```
// dispatcher.js
import { Element } from 'engine';

export default class Dispatcher extends Element {
  dispatchNotificationEvent() {
    const event = new CustomEvent('notification', { bubbles: true, composed: true });

    this.dispatchEvent(event);
    // logs C-DISPATCHER
    console.log('In dispatcher.js: ' + event.target.tagName);
  }
}
```

The component dispatches the event and logs that `event.target.tagName` is `C-DISPATCHER`.

Here's the template for the `c-wrapper` component that wraps `c-dispatcher`.

```
<!-- wrapper.html -->
<template>
  <h1>Wrapper component with slot</h1>
  <div>
    <slot></slot>
  </div>
</template>
```

Here's the JavaScript file for `c-wrapper`.

```
// wrapper.js
import { Element } from 'engine';

export default class Wrapper extends Element {
  constructor() {
    super();
    this.template.addEventListener('notification',
      this.handleNotification.bind(this)
    );
  }

  handleNotification(evt) {
    // logs C-DISPATCHER
    console.log('In wrapper.js: ' + evt.target.tagName); // TODO output
  }
}
```

`c-wrapper` adds an event handler using `addEventListener()` and implements the `handleNotification()` handler. We add the handler in code so that we can log the output of `evt.target.tagName`.

Here's the template for the outermost component, `c-handler`, which adds a handler in its markup.

```
<!-- handler.html -->
<template>
  <p>Outermost Handler component</p>
```



```

    <c-wrapper>
      <c-dispatcher onnotification={handleNotification}></c-dispatcher>
    </c-wrapper>
  </template>

```

Here's the JavaScript file for `c-handler`.

```

// handler.js
import { Element } from 'engine';

export default class Handler extends Element {
  handleNotification(evt) {
    // logs C-WRAPPER
    console.log('In handler.js: ' + evt.target.tagName);
  }
}

```

The `c-handler` component logs a different value for `evt.target.tagName`. Remember that the target value of the event changes to the current custom element (component) as the event bubbles up.

Listen for Changes to Input Fields

To listen for changes from an element in your template that accepts input, such as a text field (`<input>` or `<lightning-input>`), use the `onchange` event with an `@track` property.

```

<!-- form.html -->
<template>
  <input type="text" value={myValue} onchange={handleChange} />
</template>

// form.js
export default class Handler extends Element {
  @track myValue = "initial value";

  handleChange(evt) {
    console.log('Current value of the input: ' + evt.target.value);
  }
}

```

In this example, the `handleChange()` method in the JavaScript file is invoked every time the value of the input changes.

The `myValue` property represents the value for the input element. This *property* value is not updated automatically on every change.

You might want extra validation of the value the user enters for auto-correction or restriction of some values as they type. To keep `myValue` synchronized with the current value of the input, update `myValue` in the `handleChange()` method. The following auto-corrects the typed value by removing white spaces at the beginning and end of the string. Use `evt.target.value` to get the current value of the input field.

```


// form.js
export default class Handler extends Element {
  @track myValue = "initial value";

  handleChange(evt) {
    const typedValue = evt.target.value;
    const trimmedValue = typedValue.trim(); // trims the value entered by the user
  }
}

```

```
    if (typedValue !== trimmedValue) {  
        evt.target.value = trimmedValue;  
    }  
    this.myValue = trimmedValue; // updates the internal state  
}
```

This example shows how to reset the input value property to the trimmed value in line `evt.target.value = trimmedValue`. It also shows how to keep `myValue` property in sync with the normalized value in case the component gets rehydrated (loaded with new values) in the future.

 **Note:** Mutating properties from elements defined through a template could have undesired side effects elsewhere in the component. In our example, the input value property of the element changes from the value defined in the template represented by `evt.target`. The example changes the tracked value used in the template (`myValue`) to keep the state of the component elements in sync. Otherwise, template rehydration detects a mismatch of the property value while attempting to reconcile the state of the input element with the state of your component. This mismatch generates a runtime warning, and you need to adapt your component or JavaScript to keep the integrity of the data throughout your template.


Global Publisher-Subscriber Mechanism

During the pilot, we're working out the details for the best pattern for a global publisher-subscriber mechanism.

 **Warning:** Using events as a global publisher-subscriber mechanism—for example, by adding listeners to the `window` object—is discouraged. There's no guarantee of completeness because any element in the tree can listen, stop, and retarget your event.

Removing Event Listeners

The framework takes care of managing and cleaning up all of the listeners for you, as part of the component lifecycle, so you don't have to worry about it.

 **Warning:** If, for some reason, you need to add a listener to the global `window` object, you're responsible for removing the listener yourself within the appropriate lifecycle hook. In those cases, use the `connectedCallback` and `disconnectedCallback` methods to add and remove the event listener respectively.

SEE ALSO:

[Shadow DOM](#)

Understanding Event Propagation

Events fired by your Lightning web components are standard DOM events. Lightning web component events propagate according to the same rules as DOM events. Understanding the behavior of event propagation lets you design efficient, maintainable event systems for your components and apps.

When designing methods for your components to communicate with each other, there's a lot to know. However, these high-level rules of thumb are easy to understand.

- **"Down and In":** To communicate with child components—components that are inside your component—set attributes on those components, or call their public methods.
- **"Up and Out":** To communicate with parent components—components that your component is inside—fire events.

Here we cover details of how your events propagate once you've fired them, so that you can better understand where those events can and can't be handled.

Event Propagation Settings

Define event propagation behavior using two properties you set on the event when you create it. These properties are explained in [Creating an Event](#). There are three possible property combinations you can use in Lightning web components.

Event Properties	Propagation Behavior
<code>{ bubbles: false, composed: false }</code>	Your component fires and handles the event itself. The event never leaves your component. This behavior is the default.
<code>{ bubbles: true, composed: true }</code>	Your component fires the event for a parent to handle in the "bubble" phase. The event propagates "up and out" from your component. This behavior is the most common, "normal" event behavior.
<code>{ bubbles: false, composed: true }</code>	Your component fires the event for a parent to handle in the "capture" phase. The event propagates "up and out" from your component. This behavior is rarely used, and should be considered an anti-pattern unless you really know what you're doing.



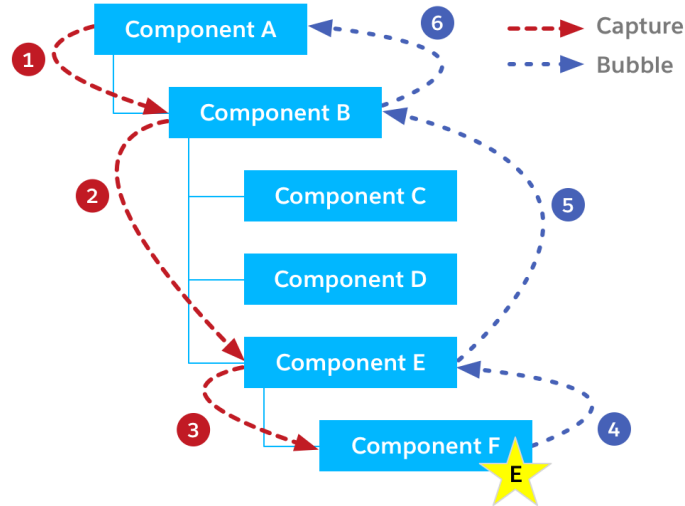
Note: The DOM standard for events (and simple math) allows for four combinations of these two properties. However, the fourth combination (`{ bubbles: true, composed: false }`) isn't used or permitted by Lightning web components, for various reasons.

Event Propagation Phases

DOM events "travel" through two consecutive propagation phases: *capture* (rarely used) and *bubble* (widely used). First, during the capture phase, events move down the DOM tree, from the top down to the element that fired the event. Second, during the bubble phase, events move back up the DOM tree, retracing their steps.

This diagram illustrates the two event propagation phases that occur along the component tree after Component F fires event E.

DOM Event Propagation Phases



There are two important things to notice.

- First, the event propagation doesn't begin with Component F. The capture phase starts at the top of the DOM tree (1).
- Second, the event doesn't visit every node in the tree. Instead, it passes through only components that contain the component that originally fired the event. In the diagram, Component C and Component D are left out, and can't observe or act upon the event.

In the absence of other influences, an event travels all the way down the component hierarchy, and then all the way back up, visiting each DOM element on the path through the hierarchy. Each element along the path that component can listen for the event, and then handle the event or stop further propagation.

In practice, when your component fires an event configured for the bubble phase (`{ bubbles: true, composed: true }`), you can ignore the capture phase (steps 1–3). Simply think of the event's path as starting with your component, and then moving to its parent, and then grandparent, and so on (steps 4–6).

 **Note:** Dispatching events or adding listeners to the capture phase is not supported.

Beyond the Basics

If you've developed Aura components, you're probably wondering about application events versus component events, container components, and other details not discussed in the preceding sections. The short answer is that these differences from standard DOM events have been eliminated, in favor of simplicity and the use of the more familiar DOM event mode.

SEE ALSO:

[JavaScript.info: Introduction to Events: Bubbling and capturing](#)

[Shadow DOM](#)

Sending Events to an Enclosing Aura Component

Lightning web components fire DOM events. An enclosing Aura component can listen for these events, just like an enclosing Lightning web component can. The enclosing Aura component can capture the event and handle it. Optionally, the Aura component can fire an Aura event to communicate with other Aura components or with the app container.

The following example illustrates this technique. This Lightning web component fires a custom `notification` event in its JavaScript file.

```
// catchandrelease.js

import { Element } from "engine";

export default class CatchAndRelease extends Element {

  /**
   * Handler for 'Fire My Toast' button.
   * @param {Event} evt click event.
   */
  handleFireMyToast(evt) {
    const eventName = 'notification';
    const event = new CustomEvent(eventName, {
      bubbles: true,
      cancelable: true,
      composed: true,
      detail: { message: 'See you on the other side.' },
    });
    this.dispatchEvent(event);
  }
}
```

The enclosing Aura component wrapper adds a handler for the custom event. Notice that the event handler, `onnotification`, matches the event name with “on” prefixed to it. That is, use `onnotification` to handle the event named `notification`.

```
<!-- catchAndReleaseWrapper.cmp -->

<aura:component implements="force:appHostable">
  <c:catchandrelease onnotification="{!c.handleCustomEvent}"/>
</aura:component>
```

 **Note:** You can only specify an `onnotification` handler in the first Aura component that the DOM event bubbles to.

The Aura component’s controller receives the event. You can handle the event however you’d like. You can optionally fire a new Aura event to communicate with other Aura components.

```
// catchAndReleaseWrapperController.js

({
  handleCustomEvent: function(cmp, evt) {

    // Get details from the DOM event fired by the Lightning web component
    var eventName = evt.getName();
    var msg = evt.getParam('message') || '';

    // This is a temporary way to support standard container events
    // Turn around and fire an Aura event with data from the DOM event
    // $A.get('e.force:standardEventNameHere')
    //     .setParams({
    //       'title': 'Event from Lightning Web Component: ' + eventName,
    //       'message': msg
    //     })
    //     .fire();
  }
});
```

```
        //    })  
        //    .fire();  
    }  
})
```

Replace `standardEventNameHere` with the event relevant to your use case. Use `setParams()` to set the attributes allowed for the event. The placeholder code is commented out.



Note: During the pilot, you can use this technique to fire events, such as `force:createRecord`, that are handled by the Lightning Experience container. However, it's an anti-pattern that clutters up your code with repetitive, boilerplate code. It's useful during the pilot because not all of the standard events are supported yet. But do plan to clean up any code that uses this technique, once Lightning web components support more events.

CHAPTER 4 Work with Salesforce Data

In this chapter ...

- [Use the Wire Service to Get Data](#)
- [Wire Service Example: Get Record Data](#)
- [User Interface API Wire Adapters and JavaScript Functions](#)
- [Call Apex Methods](#)

To create and update Salesforce data from a Lightning web component, use JavaScript APIs. To read Salesforce data, use the JavaScript `@wire` decorator to specify a data provider and tell the reactive wire service to provision data to the component.

Use the Wire Service to Get Data

To read Salesforce data, Lightning web components use a reactive wire service. Components use `@wire` in their JavaScript class to declare a data provider. A data provider is an API from a wire adapter module or a method from an Apex class.

The wire service provisions an immutable stream of data to the component. Each value in the stream is a newer version of the value that precedes it.

We call the wire service reactive in part because it supports reactive variables, which are prefixed with `$`. If a reactive variable changes, the wire service provisions new data. We say “provisions” instead of “requests” or “fetches” because if the data exists in the client cache, a network request may not be involved.



Tip: The wire service delegates control flow to the Lightning Web Components engine. Delegating control is great for read operations, but it isn't great for create, update, and delete operations. As a developer, you want complete control over operations that change data. That's why you perform create, update, and delete operations with a [JavaScript API](#) instead of with the wire service.

Wire Service Syntax

To get data, decorate a property or function with `@wire` and specify a wire adapter. Each wire adapter defines a data type.

```
import { adapterId } from 'adapter-module';  
@wire(adapterId, adapterConfig)  
propertyOrFunction;
```

- **adapterId** (Identifier)—The identifier of the wire adapter.
- **adapter-module** (String)—The identifier of the module that contains the wire adapter function.
- **adapterConfig** (Object)—A configuration object specific to the wire adapter. Configuration object property values can be strings. Wire adapters in the `lightning-ui-api-*` namespace also accept [imported references](#).
- **propertyOrFunction**—A private property or function that receives the stream of data from the wire service. If a property is decorated with `@wire`, the results are returned to the property's `data` property or `error` property. If a function is decorated with `@wire`, the results are returned in an object with a `data` property and an `error` property.

Import References to Salesforce Objects and Fields

If you're using a wire adapter in the `lightning-ui-api-*` namespace, we strongly recommend importing references to objects and fields. Salesforce verifies that the objects and fields exist, prevents objects and fields from being deleted, and cascades any renamed objects and fields into your component's source code. Importing references to objects and fields ensures that your code works, even when object and field names change.



Note: Currently, name changes don't cascade thoroughly into source code for approximately two hours. For a component used in AppBuilder, this timing is also important if the component's meta.xml file uses `<Objects>` to constrain the object home or record home.

To access object and field API names, use an `import` statement. All object and field imports come from `@salesforce/schema` [scoped packages](#).

To import a reference to an object, use this syntax.

```
import objectName from '@salesforce/schema/object';
```



```
import MyObject__c from '@salesforce/schema/MyObject__c';
import Account from '@salesforce/schema/Account';
```

To import a reference to a field, use this syntax.

```
import fieldName from '@salesforce/schema/object.field';

import MyField__c from '@salesforce/schema/MyObject__c.MyField__c';
import Name from '@salesforce/schema/Account.Name'
```

To import a reference to a field via a relationship, use this syntax. You can specify up to 5 levels of spanning fields.

```
import spanningFieldName from '@salesforce/schema/object.relationship.field';

import ParentTitleField__c from '@salesforce/schema/MyObject__c.Parent__r.Title__c';
import OwnerName from '@salesforce/schema/Account.Owner.Name';
```

This code imports the `Account.Name` field and uses it in a wire adapter's configuration object.

```
/* record.js */
import { Element, api, wire } from 'engine';
import { getRecord } from 'lightning-ui-api-record';
import NameField from '@salesforce/schema/Account.Name';

export default class Record extends Element {
  @api recordId;

  @wire(getRecord, { recordId: '$recordId', fields: [NameField] })
  record;
}
```

This code is almost identical, but it uses a string to identify the `Account.Name` field. This code doesn't get the benefits that you get from importing a reference to the field.

```
/* record.js */
import { Element, api, wire } from 'engine';
import { getRecord } from 'lightning-ui-api-record';

export default class Record extends Element {
  @api recordId;

  @wire(getRecord, { recordId: '$recordId', fields: ['Account.Name'] })
  record;
}
```

To sum it up, we recommend importing objects and fields. If your component uses `objectApiName` or `fieldApiName`, import the shape.

The only case that requires strings instead of imported objects and fields is if a component isn't aware of which object it's using. If the component is dynamic at run time, use `@wire(getObjectInfo, { objectApiName: 'Account' })` to return the object's fields. All wire adapters in the `lightning-ui-api-*` namespace respect object CRUD rules, field-level security, and sharing. If a user doesn't have access to a field, it isn't included in the response.



Tip: To check out some code, see the `account_creator` component in the [sfdx-lwc-samples](#) repo.

Mark a Configuration Object Property as Dynamic and Reactive

In the wire adapter's configuration object, prefix a property with `$` to reference an attribute on the component instance. The `$` prefix tells the wire service to treat it as a property of the class and evaluate it as `this.propertyName`. The property is reactive. If the property's value changes, new data is provisioned and the component rerenders.

In this example, `$recordId` is dynamic and reactive.

```
/* record.js */
import { Element, api, wire } from 'engine';
import { getRecord } from 'lightning-ui-api-record';
import NameField from '@salesforce/schema/Account.Name';

export default class Record extends Element {
  @api recordId;

  @wire(getRecord, { recordId: '$recordId', fields: [NameField] })
  record;
}
```

A value without a `$`, like the value of the `fields` property in the example, is a static value, in this case, `[NameField]`.

You can designate these types of configuration object properties as dynamic and reactive.

- Private properties
- Properties defined by a getter-setter pair
- Properties decorated with `@api`

Decorate a Property with `@wire`

Wiring a property is useful when you want to consume the data or error as-is.

If the property decorated with `@wire` is used as an attribute in the template and its value changes, the wire service provisions the data and triggers the component to rerender. The property is private, but reactive, like a property decorated with `@track`.

This code applies `@wire` to a property.

```
/* record.js */
import { Element, api, wire } from 'engine';
import { getRecord } from 'lightning-ui-api-record';
import NameField from '@salesforce/schema/Account.Name';

export default class Record extends Element {
  @api recordId;

  @wire(getRecord, { recordId: '$recordId', fields: [NameField] })
  record;
}
```


The property is assigned a value after component construction and before any other [lifecycle event](#). Therefore, you can access the property's value in any function, including functions used by the template or used as part of the component's lifecycle.

The object supplied to the property (in this example, `record`) has this shape.

- `data` (Any type)—The value supplied by the adapter.
- `error` (Error)—An error if the adapter wasn't able to supply the requested data or if the adapter wasn't found. Otherwise, this property is undefined.

When data becomes available from the wire adapter, it's set in the `data` property (`error` remains undefined). When newer versions of the data are available, `data` is updated.

If an error occurs in the adapter, for example when retrieving the data, `error` is populated with an error object (`data` is set to undefined).

 **Note:** When a component is rendered, but before data is supplied, the `data` and `error` properties are undefined.

Decorate a Function with `@wire`

Wiring a function is useful to perform logic whenever new data is provided or when an error occurs. The wire service provisions the function an object with `error` and `data` properties, just like a wired property.

The function is invoked whenever a value is available, which can be before or after the component is connected or rendered.

Here's our `record` component, modified to use a wired function instead of a wired property.

```
/* record.js */

import { Element, api, wire, track } from 'engine';
import { getRecord } from 'lightning-ui-api-record';
import NameField from '@salesforce/schema/Account.Name';

export default class Record extends Element {
  @api recordId;

  @track record;
  @track error;

  @wire(getRecord, { recordId: '$recordId', fields: [NameField] })
  wiredRecord({ error, data }) {
    if (error) {
      this.error = error;
    } else if (data) {
      this.record = data;
    }
  }
}
```

```
<template>
  <template if:true={error}>
    <p>An error occurred while fetching the record with ID: {recordId}</p>
    <p>Error Message: <code>{error.message}</code></p>
    <p>Error Code: <code>{error.errorCode}</code></p>
    <p>Error Status Code: <code>{error.statusCode}</code></p>
  </template>

  <template if:true={record}>
    <p>Record Name: {record.fields.Name.value}</p>
    <p>Record ID: {recordId}</p>
  </template>
```

```
<hr>
</template>
```

SEE ALSO:

[Wire Service Example: Get Record Data](#)

[Call Apex Methods](#)

Wire Service Example: Get Record Data

Let's create a simple component called `record` that has one attribute, `record-id`. The component uses the wire service to get record data and display the record name and ID, or an error message.

Let's nest two `record` components in the `playground` component in the `sfdx-lwc-samples` app. (The `playground` component is handy to use as a sample parent component.)

The `record` component has one attribute, `record-id`. It uses the ID to get the record's data. This example uses one valid ID and one invalid ID so we can see the different output.

To run this code in your scratch org, create an account and grab its ID to assign to `record-id`. To create an account, click the Lightning Data Service tab, enter an account name, and click Create Account.

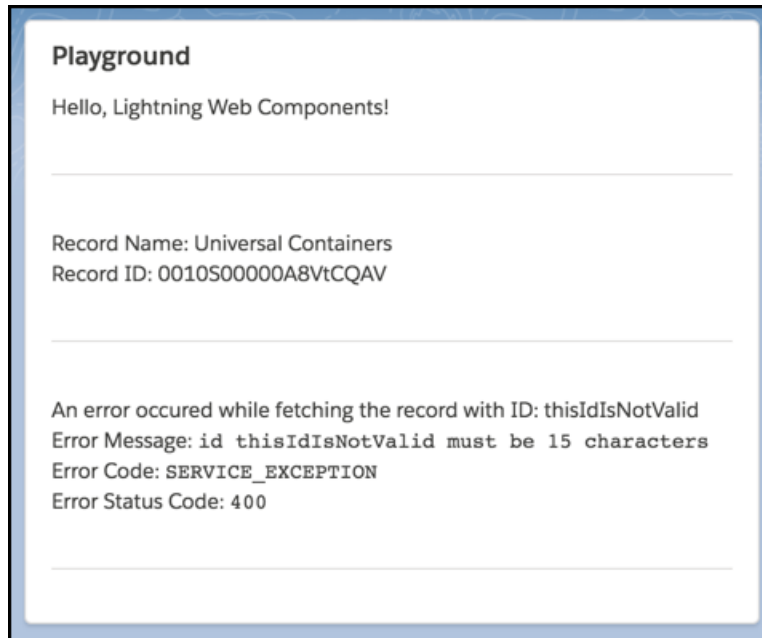
```
<!-- playground.html -->
<template>
  Hello, Lightning Web Components!
  <hr>
  <c-record record-id="0010S00000A8VtCQAV"></c-record>
  <c-record record-id="thisIdIsValidNotValid"></c-record>
</template>
```

The `playground` JavaScript file simply creates a custom element.

```
/* playground.js */
import { Element } from 'engine';

export default class Playground extends Element {}
```

To see the output, push the code to a scratch org and click the Lightning Components tab.



Now let's look at the `record` component's JavaScript.

```
/* record.js */
import { Element, api, wire } from 'engine';
import { getRecord } from 'lightning-ui-api-record';
import NameField from '@salesforce/schema/Account.Name';

export default class Record extends Element {
  @api recordId;

  @wire(getRecord, { recordId: '$recordId', fields: [NameField] })
  record;
}
```

First, the code imports the `getRecord` wire adapter. It also imports the `Account.Name` field.

The `@api` decorator makes the `recordId` property public. The parent component that consumes the `record` component sets the value on the component's `record-id` attribute. The JavaScript camel-case property `recordId` maps to the HTML kebab-case attribute, `record-id`.

The `@wire` decorator tells the wire service to use the `getRecord` wire adapter to get record data. The `getRecord` function is imported from the `lightning-ui-api-record` wire adapter module. The wire service asynchronously returns the data to the `data` and `error` objects on the decorated `record` property.

Because `$recordId` is prepended with a `$`, when its value changes, the wire service gets new data and provisions it to the component. When new data is provided, the component is rerendered.

Now let's look at the `record` component's HTML template.

The template uses the `if:true` directive to render DOM elements depending on whether the wire service returns an `error` or `data` object. To get the value of the record's `Name` field and the record's error message, traverse the returned data.

```
<!--record.html -->
<template>
  <template if:true={record.error}>
```

```

    <p>An error occurred while fetching the record with ID: {recordId}</p>
    <p>Error Message: <code>{record.error.message}</code></p>
    <p>Error Code: <code>{record.error.errorCode}</code></p>
    <p>Error Status Code: <code>{record.error.statusCode}</code></p>
  </template>

  <template if:true={record.data}>
    <p>Record Name: {record.data.fields.Name.value}</p>
    <p>Record ID: {recordId}</p>
  </template>
  <hr>
</template>

```

SEE ALSO:

[lightning-ui-api* Wire Adapters and Functions](#)

User Interface API Wire Adapters and JavaScript Functions

To work with data and metadata for Salesforce records, use wire adapters built on top of Lightning Data Service (LDS) and User Interface API. These modules use the naming format `lightning-ui-api-*`.

To get data, use `@wire` to specify a `lightning-ui-api-*` wire adapter, which provisions data. If a page is composed of components showing the same record, all components show the same version of the record. Records loaded in Lightning Data Service are cached and shared across components. Components accessing the same record see significant performance improvements, because a record is loaded once, no matter how many components are using it.

If Lightning Data Service detects a change to a record or any data or metadata it [supports](#), then all components using a relevant `@wire` receive the new value. The detection is triggered if:

- A Lightning web component mutates the record.
- The LDS cache entry expires and then a Lightning web component's `@wire` triggers a read. The cache entry and the Lightning web component must be in the same browser and app (for example Lightning Experience) for the same user.

To create and update data, use JavaScript functions exposed in the `lightning-ui-api-record` module.

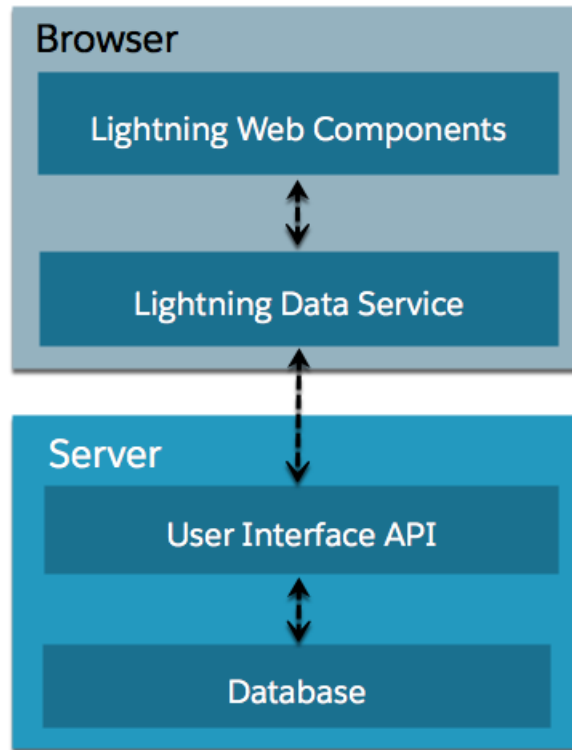
Lightning Data Service does a lot of work to make code perform well.

- Loads record data progressively.
- Caches results on the client.
- Invalidates the cache for all Salesforce data and metadata types.
- Optimizes server calls by bulkifying and deduping requests.

Lightning Data Service is built on top of [User Interface API](#). UI API is a public Salesforce API that Salesforce uses to build Lightning Experience and Salesforce for iOS and Android. Like its name suggests, UI API is designed to make it easy to build Salesforce UI.

UI API gives you data and metadata in a single response. Responses match layout changes made to the org by Salesforce admins. If an admin removes a field from a layout, that field isn't included in a response to a request for layout metadata.

UI API responses also respect CRUD access, field-level security settings, sharing settings, and perms. This respect means that the framework displays only records and fields for which users have CRUD access and FLS visibility. You get all these features by using the UI API wire adapters—`lightning-ui-api-*`.



SEE ALSO:

[lightning-ui-api* Wire Adapters and Functions](#)

Call Apex Methods

Lightning web components can import methods from Apex classes. The imported methods are functions that the component can pass to `@wire` or call imperatively.


We use the word “imperatively” in contrast with the word “declaratively.” When you call a method imperatively, you call it directly in code—`getProducts()`—and it runs.

When you program declaratively, you tell the program what to do, and it performs some additional operations under the hood. Using the wire service to call an Apex class is an example of declarative programming. You declare `@wire(getProducts)` and the wire service decides when to invoke the method. Use the wire service to call Apex methods that only get data and don’t mutate data.

To import a method from an Apex class to a Lightning web component, use an `import` statement.

```
import { apexMethod } from '@salesforce/apex/namespace.Classname.apexMethod';
```

- **namespace**—The namespace of the Salesforce organization. Specify **namespace** unless the organization uses the default namespace (`c`), in which case don’t specify it.
- **Classname**—The name of the Apex class.
- **apexMethod**—The imported symbol that identifies the Apex method.

 **Note:** During the Pilot, namespaces aren’t supported.

Expose Apex Methods to Lightning Web Components

To expose an Apex method to a Lightning web component, the method must be `static` and either `global` or `public`. Annotate the method with `@AuraEnabled`.

```
// GreetingController.cls

public class GreetingController {
    @AuraEnabled
    public static String getGreeting(String firstName) {
        return 'Hello from Apex, ' + firstName;
    }
}
```

The Lightning Component framework supports the same Apex primitive data types as the Aura framework for input and output.

The Lightning Component framework doesn't enforce any rules about the location of Apex classes. If you're using Salesforce DX, place Apex classes in the `<app dir>/main/default/classes` directory.

Enable Client-Side Caching

To improve runtime performance, set `@AuraEnabled(cacheable=true)` to cache the method results on the client. To set `cacheable=true`, a method must only get data, it can't mutate data.

Marking a method as cacheable (storable) improves your component's performance by quickly showing cached data from client-side storage without waiting for a server trip. If the cached data is stale, the framework retrieves the latest data from the server. Caching is especially beneficial for users on high latency, slow, or unreliable connections such as 3G networks.

To use the wire service to call an Apex method, you must set `cacheable=true`.

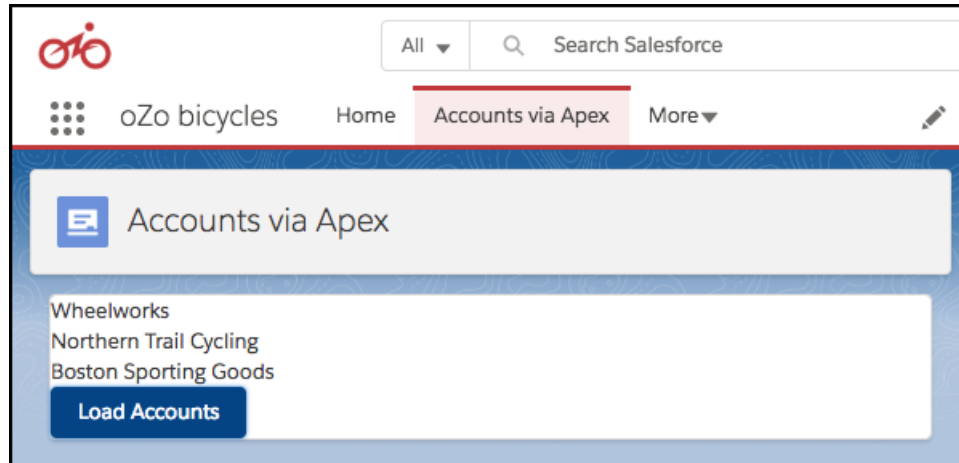
To call an Apex method imperatively, you can choose to set `cacheable=true`. This setting caches the result on the client, and prevents Data Manipulation Language (DML) operations.

The caching refresh time is the duration in seconds before an entry is refreshed in storage. The refresh time is automatically configured in Lightning Experience and the Salesforce mobile app.

Call an Apex Method Imperatively

If an Apex method mutates data and therefore isn't cacheable, you must call the method imperatively. You must also call Apex methods imperatively from modules.

Let's look at `hello_apex_accounts`, a simple component that prints a list of Accounts returned from an Apex method.



```

<!-- hello_apex_accounts.html -->
<template>
  <article class="slds-card">
    <div if:true={accounts}>
      <ul>
        <template for:each={accounts} for:item="account">
          <li key={account.id}>{account.Name}</li>
        </template>
      </ul>
    </div>
    <lightning-button variant="brand" label="Load Accounts"
      onclick={handleClick}></lightning-button>
  </article>
</template>

```

Here's the Apex class that contains the method that returns the accounts. The method is decorated with @AuraEnabled.

```

// MyAccountController.cls

public with sharing class MyAccountController {
  @AuraEnabled(cacheable=true)
  public static List<Account> getAccounts() {
    return [SELECT Id, Name FROM Account LIMIT 5];
  }
}

```

The hello_apex_accounts JavaScript code imports and calls the Apex method.

```

// hello_apex_accounts.js

import { track, Element } from 'engine';
import { getAccounts } from '@salesforce/apex/MyAccountController.getAccounts';
export default class HelloApexAccounts extends Element {
  @track accounts=[];

  async handleClick() {
    await getAccounts().then((result) => { this.accounts = result; });
  }
}

```

To use Lightning App Builder to add the component to an app, configure the `hello_apex_accounts.js-meta.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
  <apiVersion>43.0</apiVersion>
  <isExposed>true</isExposed>
  <tags>
    <tag>lightning__AppHome</tag>
  </tags>
</LightningComponentBundle>
```

Use the Wire Service to Call an Apex Method

If an Apex method is cacheable (it doesn't modify data), you can invoke it from a component via the reactive wire service.

 **Important:** Don't @wire an Apex method that creates, updates, or deletes data.

To use the wire service to invoke an Apex class, use this syntax in the component's JavaScript class.

Let's look at `hello_apex`, a simple component that prints a greeting returned from an Apex method.

```
<!-- hello_apex.html -->

<template>
  {greeting}!
</template>
```

Here's the Apex class that contains the method that returns the greeting.

```
// GreetingController.cls

public class GreetingController {
  @AuraEnabled(cacheable=true)
  public static String getGreeting(String firstName) {
    return 'Hello from Apex, ' + firstName;
  }
}
```

The `hello_world` JavaScript code imports the Apex method. It decorates `firstName` as a public property. It also designates the `$firstName` configuration object property as dynamic and reactive. If the value changes, the wire service issues a request to get more data and the component rerenders.

```
// hello_apex.js

import { api, wire, Element } from 'engine';
import { getGreeting } from '@salesforce/apex/GreetingController.getGreeting';
export default class HelloApex extends Element {
  @api firstName;

  @wire(getGreeting, {firstName: '$firstName'})
  wiredApexGreeting;

  get greeting() {
    return this.wiredApexGreeting.data || 'Hi There';
  }
}
```

```
}
```

This example shows a parent component consuming the `hello_apex` component.

```
<!-- parent_component.html -->
<template>
  <c-hello_apex first-name="Julia"></c-hello_apex>
</template>
```



Example: The github.com/forcedotcom/ebikes-lwc sample repo has several examples of calling an Apex method from a Lightning web component. Search the repo for “apex”.

SEE ALSO:

[Use the Wire Service to Get Data](#)

CHAPTER 5 Component Security with Locker Service

In this chapter ...

- [DOM Access Containment](#)
- [Secure Wrappers for Global References](#)
- [Restricted Access to Salesforce Global Variables](#)
- [Unsupported Browser APIs During Pilot](#)
- [Locker Service Disabled for Unsupported Browsers](#)

Locker Service provides component isolation and security that allows code from many sources to execute and interact using safe, standard APIs and event mechanisms. Locker Service is enabled for all custom Lightning web components.

SEE ALSO:

[DOM Access Containment](#)

DOM Access Containment

A component can only traverse the DOM and access elements that it created. This behavior prevents the anti-pattern of reaching into DOM elements owned by other components.

Lightning web components use the Shadow DOM, which is the Web Component standard that creates CSS and DOM encapsulation. Therefore, Lightning web components shouldn't use the `window` or `document` global properties to query for DOM elements. For example, use `this.root.querySelector()` instead of `document.root.querySelector()`.

For more information, see [Shadow DOM](#).

SEE ALSO:

[Component Security with Locker Service](#)

Secure Wrappers for Global References

Locker Service applies restrictions to global objects by hiding the objects or providing secure versions of those objects. For example, the secure version of `window` is `SecureWindow`. You can interact with the secure version in the same way you interact with the non-secure one, except that some methods and properties are filtered, and others are hidden.

Here's a list of the secure objects that you most commonly encounter.

SecureDocument

Secure wrapper for the `document` object, which represents the root node of the HTML document or page. The `document` object is the entry point into the page's content, which is the DOM tree.

SecureElement

Secure wrapper for the `Element` object, which represents various HTML elements. `SecureElement` is wrapped in a `Proxy` object as a performance optimization so that its data can be lazily filtered when it's accessed. Therefore, the HTML element is represented by a `Proxy` object if you're debugging in the browser console.

SecureObject

Secure wrapper for an object that is wrapped by Locker Service. When you see a `SecureObject`, it typically means that you don't have access to the underlying object and some properties aren't available.

SecureWindow

Secure wrapper for the `window` object, which represents a window containing a DOM document.

SecureDocument and SecureWindow Considerations

Shadow DOM Semantics for DOM Access

Lightning web components use the Shadow DOM, which is the Web Component standard that creates CSS and DOM encapsulation. Therefore, Lightning web components shouldn't use the `window` or `document` global properties to query for DOM elements. For example, use `this.root.querySelector()` instead of `document.root.querySelector()`.

For more information, see [Shadow DOM](#).

Shared Locker

A Lightning web component and an Aura component share the same locker if they belong to the same namespace. In other words, both types of component share the same wrapped window (`SecureWindow`) and document (`SecureDocument`) instance.

SEE ALSO:

[Component Security with Locker Service](#)

[Restricted Access to Salesforce Global Variables](#)

Restricted Access to Salesforce Global Variables

LockerService blocks access to some global JavaScript objects that are available to other Salesforce features.

Lightning web components **don't have access** to:

- `$A`
- `Aura`
- `Sfdc`
- `sforce`

SEE ALSO:

[Component Security with Locker Service](#)

[Secure Wrappers for Global References](#)

Unsupported Browser APIs During Pilot

During the pilot, there are some gaps in browser API support. We're working on closing these gaps.

During the pilot, the following browser APIs **don't work** with Lightning web components.

- `Window.getComputedStyle()`
- `MutationObserver.observe()`

During the pilot, the following APIs **don't work** with Lightning web components when `Node` is `document`, or when the argument is an element created using `document.createElement()`.

- `document.contains()`
- `Node.insertBefore()`
- `Node.removeChild()`
- `Node.replaceChild()`
- `Node.appendChild()`
- `Node.prepend()`
- `Node.append()`
- `Node.isEqualNode()`
- `Node.isSameNode()`

The following is a forward-looking statement, with all the usual caveats: Salesforce intends to close these browser API gaps before Lightning web components are Generally Available.

Locker Service Disabled for Unsupported Browsers

Locker Service relies on some JavaScript features in the browser: support for strict mode, the `Map` object, and the `Proxy` object. If a browser doesn't meet the requirements, Locker Service can't enforce all its security features and is disabled.

Locker Service is disabled for unsupported browsers. If you use an unsupported browser, you're likely to encounter issues that won't be fixed. Make your life easier and your browsing experience more secure by using a supported browser.



Note: The Locker Service requirements align with the supported browsers for Lightning Experience, except for IE11. Locker Service is disabled for IE11. We recommend using supported browsers other than IE11 for enhanced security.

CHAPTER 6 Use Lightning Web Components in Lightning Experience and Salesforce

In this chapter ...

- [View a List of Lightning Components in Your Org](#)
- [Navigate to Pages in Lightning Experience and Salesforce](#)
- [PageReference Types](#)
- [Configure a Component for Lightning Pages and Lightning App Builder](#)

Learn how to use Lightning web components in many different contexts.

View a List of Lightning Components in Your Org

From Setup, view a list of the Lightning web components and Aura components in your org.

From Setup, enter *Lightning Components* in the Quick Find box, then select **Lightning Components**. From this page you can:

- View the Lightning web components and Aura components in your org.
- View component details. Click a component name or label.
- Open an Aura component in the Developer Console. Click the component name, then click Developer Console.

Navigate to Pages in Lightning Experience and Salesforce

To navigate to a page in Lightning Experience and the Salesforce mobile app, instead of a URL, use a `PageReference`, which describes the page. Using a `PageReference` insulates your component from future changes to URL formats. It also allows your component to be used in multiple applications, each of which can use different URL formats.

A `PageReference` is a JavaScript object that describes the page type and its attributes, and the state of the page.

Name	Type	Description	Required
<code>type</code>	String	A <code>PageReference</code> object must be defined with a specific type. The type generates a unique URL format and provides attributes that apply to all pages of that type. See PageReference Types .	Yes
<code>attributes</code>	Object	Attributes that control the behavior of the page. For example, the record or object to work with.	Yes
<code>state</code>	Object	Set of key-value pairs with string keys and string values. These parameters conditionally customize content within a given page.	No

To navigate to a `PageReference`, use the [navigation service](#).

1. Import the `lightning-navigation` module.

```
import { NavigationMixin } from 'lightning-navigation';
```

2. Apply the `NavigationMixin` function to your component's base class.

```
export default class MyCustomElement extends NavigationMixin(Element) {}
```

3. Create a plain JavaScript object that defines the [page reference](#).
4. Call the navigation service's `[NavigationMixin.Navigate] (pageReference, [replace])` function to dispatch the navigation request. If `replace` is set to `true`, the `pageReference` replaces the existing entry in the browser history so the user doesn't have to press the back button twice. The default value is `false`.

```
navigateNext() {  
  this[NavigationMixin.Navigate]({  
    type: 'standard__navItemPage',  
    attributes: {  
      apiName: this.tabName,  

```

```

    },
  });
}

```

Let's look at an example.

The [sfdx-lwc-samples](#) app uses the `navtab` component at the bottom of each lesson to allow users to navigate to the next lesson.

Look at this code excerpted from the `welcome` component, which is the first page of the Lightning Web Components sample app. At the bottom of the page, there's a `navtab` component with the label **Next Lesson**.

```

<!-- welcome.html -->
<!-- lots and lots of code removed for brevity -->

<div class="slds-m-vertical_large">
  <c-navtab tab-name="lightning_components" label="Next Lesson"></c-navtab>
</div>

```

The `navtab` component customizes the `lightning-button` component to navigate to the next page in the app. The consumer specifies a `tabName` and a `label`.

```

<!-- navtab.html -->

<template>
  <lightning-button variant="brand" label={label} title={label}
onclick={navigateNext}></lightning-button>
</template>

```

```

/* navtab.js */

import { Element, api, wire } from 'engine';
import { NavigationMixin } from 'lightning-navigation';

export default class Navtab extends NavigationMixin(Element) {
  @api tabName;
  @api label;

  navigateNext() {
    this[NavigationMixin.Navigate]({
      type: 'standard__navItemPage',
      attributes: {
        apiName: this.tabName,
      },
    });
  }
}

```

Basic Navigation

The `NavigationMixin` adds two APIs to your component's class. Since these APIs are methods on the class, they must be invoked with the `this` reference.

- `[NavigationMixin.Navigate](pageReference, [replace])`

A component calls `this[NavigationMixin.Navigate]` to navigate to another page in the application.

- `[NavigationMixin.GenerateUrl]` (`pageReference`)

A component calls `this[NavigationMixin.GenerateUrl]` to get a promise that resolves to the resulting URL. The component can use the URL in the `href` attribute of an anchor. It can also use the URL to open a new window using the `window.open(url)` browser API.

In this example, the `url` comes from the navigation service. Use the Navigation API in the `click` event handler to navigate to the page and avoid reloading the entire browser web page. Assigning the URL to the `href` allows users to copy the link or to open the link in a new window.

```
<template>
  <div>
    <a href={url} onclick={handleClick}>Account Home</a>
  </div>
</template>
```

This example shows a link to Account Home. It uses `NavigationMixin` to avoid hard-coding a URL. It also uses `NavigationMixin` to trigger navigation without reloading the page.

```
import { Element, wire, track } from 'engine';
import { NavigationMixin } from 'lightning-navigation';

export default class NavigationLinkExample extends NavigationMixin(Element) {

  @track
  url;

  connectedCallback() {
    // Store the PageReference in a variable to use in handleClick.
    // This is a plain Javascript object that conforms to the
    // PageReference type by including "type" and "attributes" properties.
    // The "state" property is optional.
    this.accountHomePageRef = {
      type: "standard__objectPage",
      attributes: {
        "objectApiName": "Account",
        "actionName": "home"
      }
    };
    this[NavigationMixin.GenerateUrl](this.accountHomePageRef)
      .then(url => this.url = url);
  }

  handleClick(evt) {
    // Stop the event's default behavior.
    // Stop the event from bubbling up in the DOM.
    evt.preventDefault();
    evt.stopPropagation();
    // Navigate to the Account Home page.
    this[NavigationMixin.Navigate](this.accountHomePageRef);
  }
}
```

Change Page State

The key-value pairs of the `PageReference` state property are serialized to URL query parameters. To create a deep link that describes the page and that a user can bookmark, update the `state` property.

The component in this example has a link that updates the current page's state when clicked. When the page state changes, the property decorated with `@wire (CurrentPageReference)` is updated, which rerenders the component.

```
<template>
  <div>
    <a href={showPanelUrl} onclick={handleShowPanelClick}>Show Panel</a>
  </div>
  <div if:true={showPanel}>
    <h1>This is the panel</h1>
    <a href={noPanelUrl} onclick={handleNoPanelClick}>Hide Panel</a>
  </div>
</template>
```

```
import { Element, wire, track } from 'engine';
import { CurrentPageReference, NavigationMixin } from 'lightning-navigation';

export default class PageStateChangeExample extends NavigationMixin(Element) {

  // Injects the page reference that describes the current page
  @wire(CurrentPageReference)
  currentPageReference; // NOTE: This value is read-only

  @track
  showPanelUrl;

  @track
  noPanelUrl;

  // Determines the display for the component's panel
  get showPanel() {
    // Derive this property's value from the current page state.
    return this.currentPageReference &&
      this.currentPageReference.state.c__showPanel == "true";
  }

  // Returns a page reference that matches the current page
  // but sets the "c__showPanel" page state property to "true"
  get showPanelPageReference() {
    return getUpdatedPageReference({
      c__showPanel: "true" // Value must be a string
    });
  }

  // Returns a page reference that matches the current page
  // but removes the "c__showPanel" page state property
  get noPanelPageReference() {
    return getUpdatedPageReference({
      // This will cause this property to be removed from the state
    });
  }
}
```

```

        c__showPanel: undefined
    });
}

// Utility function that returns a copy of the current page reference
// after applying the stateChanges to the state on the new copy
getUpdatedPageReference(stateChanges) {
    // The currentPageReference property is read-only.
    // To navigate to the same page with a modified state,
    // copy the currentPageReference and modify the copy.
    return Object.assign({}, this.currentPageReference, {
        // Copy the existing page state to preserve other parameters
        // If any property on stateChanges is present but has an undefined
        // value, that property in the page state will be removed.
        state: Object.assign({}, this.currentPageReference.state, stateChanges)
    });
}

connectedCallback() {
    this[NavigationMixin.GenerateUrl](this.showPanelPageReference)
        .then(url => this.showPanelUrl = url);
    this[NavigationMixin.GenerateUrl](this.noPanelPageReference)
        .then(url => this.noPanelUrl = url);
}

handleShowPanelClick(evt) {
    evt.preventDefault();
    evt.stopPropagation();
    // This example passes true to the 'replace' argument on the navigate API
    // to change the page state without pushing a new history entry onto the
    // browser history stack. This prevents the user from having to press back
    // twice to return to the previous page.
    this[NavigationMixin.Navigate](this.showPanelPageReference, true);
}

handleNoPanelClick(evt) {
    evt.preventDefault();
    evt.stopPropagation();
    this[NavigationMixin.Navigate](this.noPanelPageReference, true);
}
}

```

The PageReference object is frozen, which means that you can't change it directly. To navigate to the same page with a modified state, copy the current PageReference and modify the copy using `Object.assign({}, pageReference)`.

- state properties must use a namespace prefix followed by two underscores, `__`. If the component isn't part of a managed package, use `c` for the namespace prefix. If the component is part of a managed package, use the package's namespace.
- Since the key-value pairs of `PageReference.state` are serialized to URL query parameters, all the values must be strings.
- Code that consumes values from state must parse the value into its proper format.
- To delete a value from the state object, define it as `undefined`.

Navigate to Different Page Types

The navigation service supports different kinds of pages in Lightning. Each [page reference type](#) supports a different set of attributes and state properties. Both APIs support these page reference types.

This code shows examples of navigating to different types of pages in Lightning. These examples show how to create page reference objects of different types and navigate to those pages. You can also pass these page references to the `[NavigationMixin.GenerateUrl](pageReference)` API to retrieve a `promise` that resolves to a URL for the page.

```
import { Element, wire } from 'engine';
import { NavigationMixin } from 'lightning-navigation';

export default class NavigationToPagesExample extends NavigationMixin(Element) {

  navigateToObjectHome() {
    // Navigate to the Case object home page.
    this[NavigationMixin.Navigate]({
      type: 'standard__objectPage',
      attributes: {
        objectApiName: 'Case',
        actionName: 'home'
      }
    });
  }

  navigateToListView() {
    // Navigate to the Contact object's Recent list view.
    this[NavigationMixin.Navigate]({
      type: 'standard__objectPage',
      attributes: {
        objectApiName: 'Contact',
        actionName: 'list'
      },
      state: {
        // 'filterName' is a property on the page 'state'
        // and identifies the target list view.
        // It may also be an 18 character list view id.
        filterName: 'Recent' // or by 18 char "00BT00000002TONQMA4"
      }
    });
  }

  navigateToNewRecordPage() {
    // Navigate to the new Account record page
    // to create a new Account.
    this[NavigationMixin.Navigate]({
      type: 'standard__objectPage',
      attributes: {
        objectApiName: 'Account',
        actionName: 'new'
      }
    });
  }
}
```

```
navigateToRecordViewPage() {
    // View a custom object record.
    this[NavigationMixin.Navigate]({
        type: 'standard__recordPage',
        attributes: {
            recordId: "a03B00000002tEurIAE",
            objectApiName: 'namespace__ObjectName', // objectApiName is optional
            actionName: 'view'
        }
    });
}

navigateToRecordEditPage() {
    // Navigate to the Account record page
    // to view a particular record.
    this[NavigationMixin.Navigate]({
        type: 'standard__recordPage',
        attributes: {
            recordId: "001B0000000ZBz22IAD",
            objectApiName: 'Account', // objectApiName is optional
            actionName: 'edit'
        }
    });
}

navigateToRelatedList() {
    // Navigate to the CaseComments related list page
    // for a specific Case record.
    this[NavigationMixin.Navigate]({
        type: 'standard__recordRelationshipPage',
        attributes: {
            recordId: '500xx000000Ykt4AAC',
            objectApiName: 'Case',
            relationshipApiName: 'CaseComments',
            actionName: 'view'
        }
    });
}

navigateToTabPage() {
    // Navigate to a specific CustomTab.
    this[NavigationMixin.Navigate]({
        type: 'standard__navItemPage',
        attributes: {
            // CustomTabs from managed packages are identified by their
            // namespace prefix followed by two underscores followed by the
            // developer name. E.g. 'namespace__TabName'
            apiName: 'CustomTabName'
        }
    });
}
```

```
}  
}
```

SEE ALSO:

[Lightning Web Component Library: lightning-navigation](#)

PageReference Types

A `PageReference` object must be defined with a specific type. The type generates a unique URL format and provides attributes that apply to all pages of that type.

The following types are supported.

- Lightning Component
- Knowledge Article
- Named Page
- Navigation Item Page
- Object Page
- Record Page
- Record Relationship Page



Note: `PageReference` objects are not supported in Communities and Lightning Out.

Lightning Component Type

A Lightning component. To make an addressable Lightning web component, embed it in an Aura component that implements the `lightning:isUrlAddressable` interface.

Type

```
standard__component
```

Attributes

Attribute	Type	Description	Required
<code>componentName</code>	String	The Lightning component name in the format <code>namespace__componentName</code> .	Yes

Example

```
{  
  "type": "standard__component",  
  "attributes": {  
    "componentName": "c__MyLightningComponent"  
  },  
  "state": {  
    "c__counter": "5"  
  }  
}
```



```
}  
}
```

You can pass any key and value in the `state` object. The key must include a namespace, and the value must be a string.

Knowledge Article Page Type

A page that interacts with a Knowledge Article record.

Type

```
standard__knowledgeArticlePage
```

Properties

Property	Type	Description	Required
<code>articleType</code>	String	The <code>ArticleType</code> API name of the Knowledge Article record.	Yes
<code>urlName</code>	String	The value of the <code>urlName</code> field on the target <code>KnowledgeArticleVersion</code> record. The <code>urlName</code> is the article's URL.	Yes

Example

```
{  
  "type": "standard__knowledgeArticlePage",  
  "attributes": {  
    "articleType": "Briefings",  
    "urlName": "February-2017"  
  }  
}
```

Named Page Type

A standard page with a unique name. Only `home`, `chatter`, `today`, and `dataAssessment` are supported.

Type

```
standard__namedPage
```

Properties

Property	Type	Description	Required
<code>pageName</code>	String	The unique name of the page.	Yes

Example

```
{  
  "type": "standard__namedPage",  
  "attributes": {  
    "pageName": "home"  
  }  
}
```

```
    }
}
```

Navigation Item Page Type

A page that displays the content mapped to a CustomTab. Visualforce tabs, Web tabs, Lightning Pages, and Lightning Component tabs are supported.

Type

```
standard__navItemPage
```

Properties

Property	Type	Description	Required
apiName	String	The unique name of the CustomTab.	Yes

Example

```
{
  "type": "standard__navItemPage",
  "attributes": {
    "apiName": "MyCustomTabName"
  }
}
```

Object Page Type

A page that interacts with a standard or custom object in the org and supports standard actions for that object.

Type

```
standard__objectPage
```

Properties

Property	Type	Description	Required
actionName	String	The action name to invoke. Valid values include <code>home</code> , <code>list</code> , and <code>new</code> .	Yes
objectApiName	String	The API name of the standard or custom object. For custom objects that are part of a managed package, prefix the custom object with <code>ns__</code> .	Yes

Standard Object Example

```
{
  "type": "standard__objectPage",
  "attributes": {
    "objectApiName": "Case",
    "actionName": "home"
  }
}
```

```
    }
}
```

Navigate to a Specific List View Example

```
{
  "type": "standard__objectPage",
  "attributes": {
    "objectApiName": "ns__Widget__c",
    "actionName": "list"
  },
  "state": {
    "filterName": "Recent"
  }
}
```

Record Page Type

A page that interacts with a record in the org and supports standard actions for that record.

Type

```
standard__recordPage
```

Properties

Property	Type	Description	Required
actionName	String	The action name to invoke. Valid values include clone, edit, and view.	Yes
objectApiName	String	The API name of the record's object. Optional for lookups.	No
recordId	String	The 18 character record ID.	Yes

Example

```
{
  "type": "standard__recordPage",
  "attributes": {
    "recordId": "001xx000003DGg0AAG",
    "objectApiName": "PersonAccount",
    "actionName": "view"
  }
}
```

Record Relationship Page Type

A page that interacts with a relationship on a particular record in the org. Only related lists are supported.

Type

```
standard__recordRelationshipPage
```

Properties

Property	Type	Description	Required
actionName	String	The action name to invoke. Only <code>view</code> is supported.	Yes
objectApiName	String	The API name of the object that defines the relationship. Optional for lookups.	No
recordId	String	The 18 character record ID of the record that defines the relationship.	Yes
relationshipApiName	String	The API name of the object's relationship field	No

Example

```
{
  "type": "standard__recordRelationshipPage",
  "attributes": {
    "recordId": "500xx000000Ykt4AAC",
    "objectApiName": "Case",
    "relationshipApiName": "CaseComments",
    "actionName": "view"
  }
}
```

SEE ALSO:

[Lightning Web Component Library: lightning-navigation](#)

Configure a Component for Lightning Pages and Lightning App Builder

There are a few steps to take before you can use your custom Lightning web components in either Lightning pages or the Lightning App Builder.

1. Deploy My Domain in Your Org

You must deploy My Domain in your org if you want to use Lightning Web components in Lightning pages or the Lightning App Builder.

For more information about My Domain, see the [Salesforce Help](#).

2. Define Component Metadata in the Configuration File

The `<component>.js-meta.xml` file defines the metadata values for the component, including the design configuration for components intended for use in the Lightning App Builder. There are different tag sets you can use to make your component Lightning App Builder-ready.

Use the `<tags>` tag set to define what types of Lightning pages your component can be used on, and to enable your component to be assigned the ID of the current record.

Configure your component's properties and set its supported objects using the `<tagConfigs>` tag set.

For more details on these tags and their values, including a full code sample, see [Component Configuration File Tags](#).

Component Configuration File Tags

Each Lightning Web Component project folder must include a configuration file. The configuration file defines the metadata values for the component, including the design configuration for components intended for use in Lightning App Builder.

Make Your Component Width-Aware with `lightning-flexipage-service`

When you add a component to a region on a page in the Lightning App Builder, `lightning-flexipage-service` passes the region's width to the component. With `lightning-flexipage-service` and some strategic CSS, you can tell the component to render in different ways in different regions at run time.

Component Configuration File Tags

Each Lightning Web Component project folder must include a configuration file. The configuration file defines the metadata values for the component, including the design configuration for components intended for use in Lightning App Builder.

apiVersion

A double value that binds the component to a Salesforce API version.

description

A short description of the component, usually a single sentence. Appears in list views, like the list of Lightning Components in Setup, and in the Lightning App Builder.

isExposed

A Boolean value used to expose the component in all orgs. To make a component usable in a managed package or by Lightning App Builder in another org, set `isExposed` to `true`.



Note: Packaging is not supported in the Lightning Web Components pilot.

masterLabel

The title of the component. Appears in list views, like the list of Lightning Components in Setup, and in the Lightning App Builder.

tags

Specifies which types of Lightning page the component can be added to in the Lightning App Builder, and whether the component is assigned the ID of the current record. You must specify at least one Lightning page type if you want your component to appear in the Lightning App Builder.

Supports the `tag` subtag.

tag

Valid values are:

Value	Description
<code>lightning__AppHome</code>	Enables a component to be used on a Lightning page of type App Home.
<code>lightning__Home</code>	Enables a component to be used on a Lightning page of type Home.
<code>lightning__RecordHome</code>	Enables a component to be used on a Lightning page of type Record Home.
<code>lightning__HasRecordId</code>	<p>Enables a component to be assigned the ID of the current record. The current record ID is useful if the component is used on a Lightning record page, or used as an object-specific custom action, or used as an action override in Lightning Experience or the Salesforce app.</p> <p>For this tag to work properly, set <code>recordId</code> as a public property in your <code>component.js</code> file.</p> <pre>@api recordId;</pre>

Value	Description
	When your component is invoked in a record context in Lightning Experience or in the Salesforce app, the <code>recordId</code> is set to the 18-character ID of the record being viewed, for example: 001xx000003DGSWAA4.

tagConfigs

Configure the component for different page types and define component properties. For example, a component could have different properties on a record home page than on the Salesforce Home page or on an app home page.

Supports the `tagConfig` subtag.

tagConfig

Use a separate `tagConfig` for each different page type configuration. Specify one or more page types in the `tags` attribute, such as `<tagConfig tags="lightning__RecordHome">` or `<tagConfig tags="lightning__RecordHome,lightning__AppHome">`.

The `tags` attribute value that you specify must match one or more of the page types that you listed under `<tags>` for the component.

Supports the `property` and `objects` subtags.

property

Specifies a public property of a component that can be set in Lightning App Builder. The component author defines the property in the component's JavaScript class using the `@api` decorator.

The `property` tag supports these attributes:

Attribute	Description	Required
datasource	<p>Renders a field as a picklist, with static values. Supported only if the <code>type</code> attribute is <code>String</code>.</p> <pre><property name="Name" datasource="value1,value2,value3" /></pre> <p>You can also set the picklist values dynamically using an Apex class.</p> <pre><property name="Name" datasource="apex://MyCustomPickList"/></pre> <p>See Create Dynamic Picklists for Your Custom Components for more information.</p>	No
default	The default value for the attribute.	No
description	Displays as an i-bubble for the attribute in the tool.	No
label	Displays as a label for the attribute in the tool.	No
max	<p>If the <code>type</code> attribute is <code>Integer</code>, this value is the maximum allowed value.</p> <p>If the <code>type</code> attribute is <code>String</code>, this value is the maximum length allowed.</p>	No
min	<p>If the <code>type</code> attribute is <code>Integer</code>, this value is the minimum allowed value.</p> <p>If the <code>type</code> attribute is <code>String</code>, this value is the minimum length allowed.</p>	No
name	Required if you're setting properties for your component. This is the attribute name. This value must match the property name in the component's JavaScript class.	Yes

Attribute	Description	Required
<code>placeholder</code>	Input placeholder text for the attribute when it displays in the tool. This text is the ghost text in text fields and text areas before a user starts typing. Supported only if the <code>type</code> attribute is <code>String</code> .	No
<code>required</code>	Specifies whether the attribute is required. The default value is <code>false</code> .	No
<code>type</code>	Required if you're setting properties for your component. This is the attribute data type, such as <code>Integer</code> , <code>String</code> , or <code>Boolean</code> .	Yes

objects

A set of one or more objects the component is supported for. This tag set works only inside a parent `tagConfig` that's configured for `lightning__RecordHome`. Specify the `objects` tag set only once inside a `tagConfig` set. Supports the `object` subtag.

object

Defines which objects the component is supported for. Use one `object` tag for each supported object. You can't use `'*'` to denote all objects.

SEE ALSO:

[Component Configuration File](#)

Make Your Component Width-Aware with `lightning-flexipage-service`

When you add a component to a region on a page in the Lightning App Builder, `lightning-flexipage-service` passes the region's width to the component. With `lightning-flexipage-service` and some strategic CSS, you can tell the component to render in different ways in different regions at run time.

For example, the List View component renders differently in a large region than in a small region because it's a width-aware component.

The screenshot illustrates a Lightning Web Component (LWC) that uses the `lightning-flexipage-service` to adapt its layout based on the available width. The component is titled "My Opportunities" and displays a list of opportunities. In the narrow view (left), it shows a table with columns for Opportunity Name, Account Name, and Amount. In the wide view (right), it shows a detailed card for the first opportunity, "Burlington Textiles Weaving Plant Generator", with fields for Account Name, Amount, Close Date, Stage, and Opportunity Owner.

In the component's JavaScript class, import the `getRegionInfo` service. Use `@track` to mark `width` as a private reactive property. The `getRegionInfo` method returns the region's width value depending where the component is rendered.

```
// test_class.js
import { Element, track, wire } from 'engine';
import { getRegionInfo } from 'lightning-flexipage-service';

export default class TestClass extends Element {
  @track width;
  @wire(getRegionInfo, {})
  wiredFlexipageRegionInfo(value) {
    this.width = value.width;
  }
}
```

Use the private reactive `width` property in the HTML template. When the value of `width` changes, the component rerenders.

```
<!-- test_class.html -->
<template>
  <div class={width}>
    ....
  </div>
</template>
```

Use CSS to define the component's behavior when it renders in different region widths. Valid CSS class values are `SMALL`, `MEDIUM`, and `LARGE`. This sample CSS snippet tells the component to render with a red background when it's in a large region and apply a blue background when it's in a small region.

```
/* test_class.css */
div .LARGE {
  background : red;
  ....
```



```
}  
div .SMALL {  
  background : blue;  
  .....  
}
```

SEE ALSO:

[Lightning Web Component Library: lightning-flexipage-service](#)

CHAPTER 7 Lightning Web Components and Aura Components Working Together

In this chapter ...

- [Component Naming Schemes](#)
- [Understanding Aura Component Facets](#)
- [Lightning Web Components Inherit Aura Component Styling](#)
- [Share JavaScript Code in Lightning Web Components and Aura Components](#)

An interoperability layer enables Lightning web components and Aura components to work together in an app.

You can write new Lightning web components and add them to apps that contain Aura components. Or, you can iteratively migrate on your schedule by replacing individual Aura components in your app with Lightning web components.

Lightning web components and Aura components can work together, but there are limitations that you must understand.



Important: Aura components can contain Lightning web components. However, the opposite doesn't apply. Lightning web components can't contain Aura components.

Component Naming Schemes


The Aura model and the Lightning Web Components model each have a different syntax for references to components in markup. With the Lightning Web Components model, we use the standards set by Web Components, wherever possible.

Aura components use camel case with a colon separating the namespace and the component name.

```
<!-- In an Aura component .cmp file -->
<lightning:progressBar value="10" />
```

Lightning web components use dash-delimited, also known as kebab-case, names. Think of the dashes as the kebab skewer while the words are the ingredients on the skewer.


```
<!-- In a Lightning web component .html file -->
<lightning-progress-bar value="10"></lightning-progress-bar>
```

 **Note:** Base Lightning components in the `lightning` namespace can include a hyphen in their component names. Lightning web components that you create can't currently include a hyphen in their names. We intend to eventually support hyphens in custom component names for consistency with HTML standards. However, we still have work to do to make that happen.

Use the correct syntax for the context.

- In Aura components, use camel case: `lightning:progressBar`.
- In web components, use kebab-case: `lightning-progress-bar`.

In Aura, the interoperability layer maps the `lightning:progressBar` syntax to the underlying `lightning-progress-bar` web component.

 **Note:** You can't create a Lightning web component with a name that collides with a custom Aura component in the same namespace. For example, if you have an Aura component named `c:progressBar`, you can't create a Lightning web component named `c-progress_bar`.

Understanding Aura Component Facets

Before we dig in further, we need to understand the concept of facets in the Aura development model.

In Aura components and Lightning web components, you can add components within the body of another component. This component composition enables you to build complex components from simpler building block components.

In Aura, the `body` attribute is a facet. A facet is any attribute of type `Aura.Component[]`. This is just a fancy way of saying you can set an array of components for the attribute.

In even less fancy terms, think of a facet as a sandwich. You can stuff as many ingredients (components) as you'd like into your sandwich (facet).

```
<aura:component>
  <!-- start of component body facet-->
  <p>Feeling hungry?</p>
  <c:bacon />
  <c:lettuce />
  <c:tomato />
  <!-- end of component body facet -->
</aura:component>
```

Why are we talking about facets in Aura? Well, they're important when you consider what's allowed when you compose Aura components and Lightning web components together.

Build an Aura Component or Lightning Web Component?

You're probably hungry for more information and maybe even lunch after reading about facets in Aura. Now, let's see where facets factor into whether you should build an Aura component or a Lightning web components.

If you're building a new Lightning web component that expects other sub-components in its body, those sub-components must also be built as Lightning web components. Remember that Lightning web components can't contain Aura components.

Imagine a Lightning web component that's the outermost component in a tree of nested components. Think of the Lightning web component as the biggest doll in a matryoshka doll, also known as a Russian nesting doll. A matryoshka doll contains a set of nested dolls, each containing a smaller doll. If the biggest doll is a Lightning web component, none of the smaller nested dolls can be an Aura component.

Example: Component with No Body

Consider a `lightning-progress-bar` Lightning web component that displays a progress bar. Another Lightning web component can use `lightning-progress-bar`.

```
<template>
  <lightning-progress-bar value="10"></lightning-progress-bar>
</template>
```

You can also use `lightning-progress-bar` in an Aura component.

```
<aura:component>
  <lightning:progressBar value="10"></lightning:progressBar>
</aura:component>
```

The interoperability layer maps `lightning:progressBar` to the `lightning-progress-bar` Lightning web component.

Example: Component with Body

Consider a `c-wrapper` Lightning web component that allows other components in its body.

```
<template>
  <c-wrapper>
    <lightning-progress-bar></lightning-progress-bar>
  </c-wrapper>
</template>
```

This Lightning web component references the `lightning-progress-bar` Lightning web component in its body.

`c-wrapper` can't reference an Aura component because a Lightning web component can't receive an Aura component in its body. For example, this Aura component doesn't work because a Lightning web component, `c-wrapper`, can't reference an Aura component, `ui:label`. Even though we're using the `c-wrapper` Aura syntax, the component is still the `c-wrapper` Lightning web component under the covers of the interoperability layer.

```
<aura:component>
  <c-wrapper>
    <!-- can't use an Aura component (ui:label) here -->
    <ui:label value="foo"></ui:label>
  </c-wrapper>
</aura:component>
```

The Aura component would work if you replaced `ui:label` with a Lightning web component. Remember that a Lightning web component must use Lightning web components all the way down its component sub-tree.

SEE ALSO:

[Use Slots as Placeholders](#)

Lightning Web Components Inherit Aura Component Styling

During the pilot, to work in Lightning Experience or the Salesforce mobile application, Lightning web components must be contained by an Aura component. Aura component styles cascade on Lightning web components.

However, the styles of a Lightning web component never interfere with other Lightning web components or with Aura components. Yes, you read that correctly, Lightning web component styles don't cascade to their children. Overriding styles that you don't own can create many problems, so Lightning web components don't allow it.

Web components use Shadow DOM for DOM and CSS encapsulation.

Use CSS Variables to Reference Aura Design Tokens

[Design tokens](#) are named entities that store visual design attributes, such as margins and spacing values, font sizes and families, or hex values for colors.

Design tokens must be defined in an Aura component in the same DOM as your Lightning web component. Then, the Lightning web component can use CSS variable syntax to reference the Aura component's design token.

When you reference a design token, use the `--lwc-` prefix. For example, imagine a parent Aura component with these design tokens.

```
<!-- parent_aura_component.cmp -->
<aura:tokens>
  <aura:token name="myBodyTextFontFace"
    value="'Salesforce Sans', Helvetica, Arial"/>
  <aura:token name="myBodyTextFontWeight" value="normal"/>
  <aura:token name="myBackgroundColor" value="#f4f6f9"/>
  <aura:token name="myDefaultMargin" value="6px"/>
</aura:tokens>
```

You can reference the token name as a style in your Lightning web component's CSS.

```
/* my_web_component.css */
div {
  font-family: var(--lwc-myBodyTextFontFace, "sans-serif");
}
```

Define a default “fallback” value in case the design token isn't defined. This example uses `"sans-serif"`.

You can't style the `:root` element because it breaks encapsulation (use `:host` instead).



Note: Currently, at compile time, we replace the variables with their actual values. So, references to the variables at runtime won't work. For example, you can't use the standard APIs `CSSStyleDeclaration.getPropertyValue()` or `CSSStyleDeclaration.setPropertyValue()` on the variables.

SEE ALSO:

[CSS](#)

Share JavaScript Code in Lightning Web Components and Aura Components

To share JavaScript code between Lightning web components and Aura components, put the code in an ES6 module.

To share an ES6 module:

1. Create the ES6 module in your Lightning Web Components development environment.
2. Reference the module in a Lightning web component's JavaScript file.
3. Reference the module in an Aura component.

It's that easy!

Example

Let's step through an example to see how a Lightning web component and an Aura component can both use an ES6 module.

1. To create a Salesforce DX project, an ES6 module, and use it in a Lightning web component, follow the steps in [Share JavaScript Code](#). This example adds to the code created in the `UseModule` Salesforce DX project.
2. Create a folder for a `libcallerAura` Aura component.

```
cd path/to/your/sfdx/projects/UseModule
cd force-app/main/default/aura
mkdir libcallerAura
cd libcallerAura
```

3. Create a `libcallerAura.cmp` file.

```
<aura:component>
  <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

  <p>Aura component calling the utils lib</p>

  <!-- add the lib component -->
  <c:utils aura:id="utils" />
</aura:component>
```

The `libcallerAura` Aura component includes a reference to the `c:utils` ES6 module. We added an `aura:id` so that we can get a reference to the module in JavaScript code.

4. Create a `libcallerAuraController.js` file.

```
({
  doInit: function(cmp) {
    // Call the lib here
    var libCmp = cmp.find('utils');
    var result = libCmp.isFunction(
      function() {
        console.log(" I am a function");
      }
    );
    console.log("Is it a function?: " + result);
  }
});
```

```
    }  
  })
```

The component's controller uses `cmp.find('utils')` to match the `aura:id` in markup and get a reference to the module. The `utils` ES6 module exports `isFunction()`, which the Aura component calls. The argument passed into `isFunction` is a function so `result` is set to `true`.

5. Navigate to the `libApp` Aura application.

```
cd path/to/your/sfdx/projects/UseModule  
cd force-app/main/default/aura/libApp
```

6. Edit `libApp.app` to include the `libcallerAura` Aura component.

```
<aura:application>  
  <!-- Lightning web component -->  
  <c:libcaller />  
  
  <!-- Aura component -->  
  <c:libcallerAura />  
</aura:application>
```

7. Authenticate with your dev hub, and push your project to the scratch org created in [Share JavaScript Code](#).

```
sfdx force:auth:web:login -d -a LWC-Hub  
sfdx force:source:push  
sfdx force:org:open
```

For more information on these commands, see [Create an Empty Project for Lightning Web Components](#).

8. Navigate to `https://myDomain.lightning.force.com/c/libApp.app`.
9. Click the button and confirm that the Aura component calls the function exported by the ES6 module and logs the result to the browser console.

CHAPTER 8 Migrate Aura Components to Lightning Web Components

In this chapter ...

- [Migration Strategy](#)
- [Pick a Component to Migrate](#)
- [Migrate Component Bundle Files](#)
- [Migrate Markup](#)
- [Migrate Events](#)
- [Migrate CSS](#)
- [Migrate JavaScript](#)
- [Migrate Apex](#)
- [Data Binding Behavior Differences With Aura](#)

Aura components and Lightning web components can exist in the same application. This section helps you to migrate components and apply your existing skills by mapping concepts from Aura components to Lightning web components.

Migration Strategy

The programming model for Lightning Web Components is fundamentally different than the model for Aura components. Migrating a component is not a line-by-line conversion, and it's a good opportunity to revisit your component's design. Before you migrate an Aura component, evaluate the component's attributes, interfaces, structures, patterns, and data flow.

The easiest components to migrate are simple components that only render UI. You get more gains in performance and developer productivity by migrating larger trees of components (components within components) rather than an individual component. However, it's a useful learning experience to migrate one component and see how concepts in the Aura programming model map to concepts in the Lightning Web Components programming model.

After migrating one component, you'll be in a better position to determine whether it makes sense for you and your org to:

- Undertake a larger migration effort
- Use Lightning web components for new components only
- Stick with Aura components for now

The choice is down to you and differs for everyone, depending on use cases and available resources. Whatever decision you make, migrating a component is a valuable learning exercise.

Pick a Component to Migrate

To decide whether to migrate a component, first understand how Aura components and web components work together, and then evaluate how and where the component is used.

Before you pick a component to migrate, make sure that you [understand the concept of facets in Aura components](#).

Let's review the most important facts about how Lightning web components and Aura components work together.

- Lightning web components and Aura components can co-exist in the same runtime.
- An Aura component can contain a Lightning web component.
- A Lightning web component can't contain an Aura component.
- If an Aura component has facets that contain other Aura components, you must migrate those other Aura components too.

Migrate Component Bundle Files

The component bundle file structure is different for the Aura programming model versus the Lightning Web Components programming model.

Here's how the files map between the two programming models.

Resource	Aura File	Lightning Web Components File	See Also
Markup	<code>sample.cmp</code>	<code>sample.html</code>	Component HTML File
Controller	<code>sampleController.js</code>	<code>sample.js</code>	Component JavaScript File
Helper	<code>sampleHelper.js</code>	<code>sample.js</code>	Lightning Components Developer Guide: Helpers

Resource	Aura File	Lightning Web Components File	See Also
Renderer	<code>sampleRenderer.js</code>	<code>sample.js</code>	Lightning Components Developer Guide: Renderers
CSS	<code>sample.css</code>	<code>sample.css</code>	Component CSS File
Documentation	<code>sample.auradoc</code>	Not currently available	Lightning Components Developer Guide: Component Documentation
Design	<code>sample.design</code>	<code>sample.js-meta.xml</code>	Component Configuration File
SVG	<code>sample.svg</code>	Not currently available	Lightning Components Developer Guide: Custom Icons



Note: The controller, helper, and renderer files in an Aura component map to one JavaScript file in a Lightning web component.

Migrate Markup

An Aura `.cmp` file contains markup, including Aura-specific tags. Let's look at how this markup maps to equivalent concepts in Lightning web components.

[Migrate Attributes](#)

Migrate attributes from `<aura:attribute>` tags in an Aura component to JavaScript properties in a Lightning web component.

[Migrate Iterations](#)

Migrate `<aura:iteration>` tags in an Aura component to `for:each` in a Lightning web component.

[Migrate Conditionals](#)

Migrate `<aura:if>` tags in an Aura component to `if:true` and `if:false` in a Lightning web component.

[Migrate Expressions](#)

Migrate expressions from markup in an Aura component to JavaScript in a Lightning web component.

[Migrate Initializers](#)

Replace an `init` event handler in an Aura component with the standard JavaScript `connectedCallback()` method in a Lightning web component.

[Migrate Facets](#)

Migrate a facet in an Aura component to a slot in a Lightning web component.

[Migrate Base Components](#)

Base Lightning components have a different syntax when you use them in the two programming models. Base Lightning components are building blocks that Salesforce provides in the `lightning` namespace.

[Migrate Registered Events](#)

There's no equivalent in Lightning web components for the `<aura:registerEvent>` tag in Aura component markup to register that a component can fire an event.

[Migrate Event Handlers](#)

There's no equivalent in Lightning web components for the `<aura:handler>` tag in Aura component markup that configures an event handler.

Migrate Attributes

Migrate attributes from `<aura:attribute>` tags in an Aura component to JavaScript properties in a Lightning web component.

Let's look at a `myAttribute` attribute in an Aura component.

```
<aura:component>
  <aura:attribute name="myAttribute" default="Hello" />
</aura:component>
```

In a Lightning web component, we use a JavaScript property called `myAttribute` instead.

```
import {Element, api} from 'engine';
export default class MyComponentName extends Element {
  @api myAttribute = 'Hello';
}
```

The `@api` decorator defines `myAttribute` as a public reactive property. Public properties define the public API for a component. A parent component that uses the component in its markup can access the component's public properties. Public properties are reactive. If the value of a reactive property changes, the component's template rerenders any content that references the property.

Reference the property in the component's HTML file.

```
<template>
  {myAttribute}
</template>
```



Important: A Lightning web component must fully support the converted Aura component's attributes. If it doesn't, expect errors and breakages.

SEE ALSO:

[Public Reactive Properties](#)

Migrate Iterations

Migrate `<aura:iteration>` tags in an Aura component to `for:each` in a Lightning web component.

Here's the Aura syntax.

```
<aura:iteration items="{!v.items}" itemVar="item">
  {!item}
</aura:iteration>
```

Here's the Lightning web component syntax.

```
<template for:each={items} for:item='item'>
  <p key={item.id}>{item}</p>
</template>
```

Note that you must use a `key` to assign a unique value to each item in the list. See [Render Lists](#).

Migrate Conditionals

Migrate `<aura:if>` tags in an Aura component to `if:true` and `if:false` in a Lightning web component.

Here's the syntax in an Aura component.

```
<aura:if isTrue="{!v.something}">
  <div>Conditional Code</div>
  <aura:set attribute="else">
    <div>Conditional Code</div>
  </aura:set>
</aura:if>
```

Here's the syntax in a Lightning web component.

```
<template>
  <div if:true={something}>Conditional Code</div>
  <div if:false={something}>Conditional Code</div>
</template>
```

```
import { Element } from 'engine';
export default class MyComponentName extends Element {
  something = true;
}
```

SEE ALSO:

[Render DOM Elements Conditionally](#)

Migrate Expressions

Migrate expressions from markup in an Aura component to JavaScript in a Lightning web component.

Here's the syntax in an Aura component.

```
<aura:if isTrue="{!(v.something ? v.optionA : v.optionB) }">
  <div>Conditional Code</div>
</aura:if>
```

In a Lightning web component, use `if:true` and move the expression into JavaScript. Now the code can be unit tested, which is a very, very good thing. Here's the HTML file.

```
<template>
  <div if:true={condition}>Conditional Code</div>
</template>
```



Tip: Dynamic content in a Lightning web component's HTML file doesn't have quotes around the getter reference and there's no exclamation point or value provider (`v.`) syntax. Don't use the expression syntax from Aura components even though your fingers might be used to typing it!

Here's an expression in an Aura component.

```
<aura:if isTrue="{!v.condition}">
```

Here's similar HTML in a Lightning web component.

```
<div if:true={condition}>Conditional Code</div>
```

Here's the JavaScript file.

```
import { Element } from 'engine';
export default class MyComponentName {
  get condition() {
    return something ? true : false;
  }
}
```

SEE ALSO:

[Use Getters Instead of Expressions](#)

Migrate Initializers

Replace an `init` event handler in an Aura component with the standard JavaScript `connectedCallback()` method in a Lightning web component.

We use the `init` event in an Aura component to initialize a component after component construction but before rendering.

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
```

The `doInit` function in the component's controller performs any necessary initialization.

```
((
  doInit: function(cmp) {
    // initialize component
  }
}))
```

In a Lightning web component, use `connectedCallback()` instead in the component's JavaScript file.

```
import { Element } from 'engine';
export default class MySampleInit extends Element {
  connectedCallback() {
    // initialize component
  }
}
```

SEE ALSO:

[Run Code When a Component Is Inserted or Removed from the DOM](#)

Migrate Facets

Migrate a facet in an Aura component to a slot in a Lightning web component.

A facet in an Aura component is an attribute with a type of `Aura.Component[]`. For example:

```
<aura:component >

  <aura:attribute name="firstName" type="Aura.Component[]" />
  <aura:attribute name="lastName" type="Aura.Component[]" />

  <p>First Name: {!v.firstName}</p>
```

```
<p>Last Name: {!v.lastName}</p>
</aura:component>
```

In a Lightning web component, we use slots instead of the facets. This template defines two named slots.

```
<!-- namedslots.html -->
<template>
  <p>First Name: <slot name="firstName">Unknown</slot></p>
  <p>Last Name: <slot name="lastName">Unknown</slot></p>
</template>
```

This code consumes the `<c-namedslots>` component.

```
<!-- slotswrapper.html -->
<template>
  <c-namedslots>
    <span slot="firstName">Willy</span>
    <span slot="lastName">Wonka</span>
  </c-namedslots>
</template>
```

This HTML is the rendered output.

```
<p>First Name: <span>Willy</span></p>
<p>Last Name: <span>Wonka</span></p>
```

SEE ALSO:

[Use Slots as Placeholders](#)

[Understanding Aura Component Facets](#)

Migrate Base Components

Base Lightning components have a different syntax when you use them in the two programming models. Base Lightning components are building blocks that Salesforce provides in the `lightning` namespace.

This Aura component uses the `lightning:formattedText` base component.


```
<aura:component>
  <lightning:formattedText linkify="true" value="I like salesforce" />
</aura:component>
```

To migrate this markup to a Lightning web component:

- Change the colon that separates the namespace and component name to a dash.
- Change the camel case component name (`formattedText`) to a dash-separated name (`formatted-text`).

Here's the equivalent Lightning web component.

```
<template>
  <lightning-formatted-text linkify="true" value="I like
salesforce"></lightning-formatted-text>
</template>
```

 **Note:** Base Lightning components can include a dash in the component name, such as `formatted-text`. However, you can't create a custom Lightning web component with a dash in its name due to a limitation in the Salesforce platform. It's confusing but useful to remember!

SEE ALSO:

[Lightning Components Developer Guide: Working with Base Lightning Components](#)

Migrate Registered Events

There's no equivalent in Lightning web components for the `<aura:registerEvent>` tag in Aura component markup to register that a component can fire an event.

SEE ALSO:

[Migrate Events](#)

Migrate Event Handlers


There's no equivalent in Lightning web components for the `<aura:handler>` tag in Aura component markup that configures an event handler.

SEE ALSO:

[Migrate Events](#)

Migrate Events

Migrate component events in Aura components to standard DOM events in Lightning web components.

 **Note:** There's no direct equivalent in Lightning web components for application events in Aura components. Application events can be problematic in complex apps because any component can handle the event. This pattern can lead to code that is hard to maintain. During the Lightning Web Components pilot, we're working out the best pattern to use as an alternative to application events.

Here's an overview for working with events in Lightning web components.

Create an event

Instead of the proprietary `Event` object in Aura components, use the `Event` or `CustomEvent` standard DOM objects. There's no equivalent in Lightning web components for the `<aura:registerEvent>` tag in Aura component markup to register that a component can fire an event.

Fire an event

Instead of `event.fire()` in an Aura component, use `this.dispatchEvent(myEvent)`, which is a standard DOM method, in Lightning web components.

Handle an event

An Aura component uses the `<aura:handler>` tag in markup to define a handler. Alternatively a component can declare a handler action when it references another component in its markup. This Aura component uses `c:child` in its markup and declares a `handleChildEvent` handler for the `sampleComponentEvent` that `c:child` fires.

```
<c:child sampleComponentEvent="{!c.handleChildEvent}"/>
```

A Lightning web component can similarly declare a declarative handler. The event name in the declarative handler is prefixed by "on".

```
<c-child onsampleComponentEvent={handleChildEvent}></c-child>
```

The event handler function, `handleChildEvent`, is defined in the JavaScript file of the component.

In a Lightning web component, you can also programmatically set up a handler using the standard `addEventListener()` method in the component's JavaScript file.

SEE ALSO:

[Communicate with Events](#)

Migrate CSS

Lightning web components use standard CSS syntax. Remove the proprietary `THIS` class that Aura components use.

Here's a CSS file for an Aura component.

```
.THIS .red {  
    background-color: red;  
}  
  
.THIS .blue {  
    background-color: blue;  
}  
  
.THIS .green {  
    background-color: green;  
}
```

The CSS file for the migrated Lightning web component is identical except for the removal of `THIS`.

```
.red {  
    background-color: red;  
}  
  
.blue {  
    background-color: blue;  
}  
  
.green {  
    background-color: green;  
}
```

SEE ALSO:

[Style Components with CSS](#)

Migrate JavaScript

Move JavaScript code from your client-side controller, helper, and renderer to the single JavaScript file in a Lightning web component.

A client-side controller in an Aura component is a JavaScript object in object-literal notation containing a map of name-value pairs. A JavaScript file in Lightning web components is an ES6 module so you're using standard JavaScript rather than the proprietary format used in Aura components.

This is not a line-by-line conversion, and it's a good opportunity to revisit your component's design.

To share JavaScript code between Lightning web components and Aura components, put the code in an ES6 module. For more information, see [Share JavaScript Code in Lightning Web Components and Aura Components](#).

SEE ALSO:

[Component JavaScript File](#)

Migrate Apex

Aura components and Lightning web components both use an Apex controller to read or persist Salesforce data. There are no syntax differences for the two programming models.

Lightning web components can also use the JavaScript `@wire` decorator to wrap calls to an Apex method or the User Interface API.

SEE ALSO:

[Work with Salesforce Data](#)

Data Binding Behavior Differences With Aura

When you add a component in markup, you can initialize public property values in the component based on property values of the owner component. In Lightning Web Components, the data binding for property values is one-way. The data-binding behavior is different for the Lightning Web Components and Aura development models.

Aura

Aura has two forms of expression syntax for data binding.

Here's a summary of the differences between the forms of expression syntax.

{#expression} (Unbound Expressions)

Data updates behave as you would expect in JavaScript. Primitives, such as `String`, are passed by value, and data updates for the expression in the parent and child are decoupled.

Objects, such as `Array` or `Map`, are passed by reference, so changes to the data in the child propagate to the parent. However, change handlers in the parent aren't notified. The same behavior applies for changes in the parent propagating to the child.

{!expression} (Bound Expressions)

Data updates in either component are reflected through bidirectional data binding in both components. Similarly, change handlers are triggered in both the parent and child components.

Lightning Web Components

The data binding between components for property values is one-way.

To communicate down from a parent component to a child component, set a property or call a method on the child component.

To communicate up from a child component to a parent component, send an event.

For more information, see [Data Binding Between Components](#).

CHAPTER 9 Test Lightning Web Components

In this chapter ...

- [Tools for Testing Lightning Web Components](#)
- [Unit Test Lightning Web Components with Jest](#)

Test your Lightning web components to verify that components behave as intended. The Lightning Web Components development model provides support for multiple testing strategies and tools.

Automated tests are the best way to achieve predictable, repeatable assessments of the quality of your custom code. Writing automated tests for your custom components gives you confidence that they work as designed, and allows you to evaluate the impact of changes, such as refactoring, or of new versions of Salesforce or third-party JavaScript libraries.

Tools for Testing Lightning Web Components

Lightning Web Components and the Lightning Testing Service provide tools for verifying intended behavior. Before you start writing tests (or components!), start by thinking about what to test and how to test the components that you create.

Lightning Testing Service, or LTS, provides multiple ways for you to write tests for your components. To learn more about LTS and how to use it, see [Testing Components with Lightning Testing Service](#). As we transition LTS to support Lightning web components, we recommend that you use Jest unit tests.

- Use Jest to write tests that run at the command line or (with some configuration) within your IDE. Jest tests don't run in a browser or connect to an org, so they run fast, and give you immediate feedback while you're coding. Jest tests work with Lightning web components, but can't be used to test Aura components.
- Use Jasmine, Mocha, or (with adaptation) your favorite JavaScript test framework to write tests that run in a browser connected to a scratch org. These tests don't run with the speed of Jest tests, but they're still easy to run regularly while coding. Jasmine and Mocha tests, when run using LTS, can test Aura components and Lightning web components wrapped in Aura components. See [Create a Hello World Lightning Web Component](#) for a tutorial that includes wrapping a Lightning web component in an Aura component to display it in the browser..

In addition to these tools we've integrated with, you can also test application level features, such as page-by-page navigation, using tools such as [Selenium WebDriver](#). We don't provide a Salesforce-specific version of these tools, but as tools designed for testing web apps, they work fine with Salesforce. You can get started with these tools easily by visiting [Selenium HQ](#), or by searching for a good WebDriver tutorial.

Choose the Right Tool for the Job

Testing strategies vary with taste and organization standards, and you can choose the tools you prefer or require. That said, different tools have different strengths and weaknesses. Make sure that you understand them when developing your testing strategy.

Jest Unit Tests

Jest can be used to test Lightning web components only. They're good for:

- Testing a component in isolation
- Testing a component's public API (@api properties and methods, events)
- Testing basic user interaction (clicks)
- Verifying the DOM output of a component
- Verifying events fire when expected
- Continuous testing while coding

However, Jest tests aren't so good for:

- Testing interaction between multiple components
- Testing more complex user interaction (drag-and-drop, scrolling, and so on)
- Testing a component with numerous external dependencies (things that need to be mocked)
- Verifying how other components handle events that are fired by the component under test
- Browser compatibility testing

Mocha or Jasmine Unit Tests

Mocha and Jasmine can be used to test both Lightning web components and Aura components. They're good for:

- Testing interaction between multiple components
- Testing components in a real browser

- Testing components in a real app container
- Testing components with dependencies only available at runtime (in a real app), like static resources

However, they're not so good for:

- Anything that can be tested using Jest
- Complex flows that simulate multiple step user interactions (navigating around app, and so on)

In short, our recommendation is that you first write Jest tests for every Lightning web component. Then, if there are more behaviors remaining to test, write Jasmine or Mocha tests to cover those use cases. Finally, for higher level, complex flow testing, investigate using WebDriver or a similar tool.

Unit Test Lightning Web Components with Jest

Use Jest to write unit tests for all of your Lightning web components. Jest is a powerful tool with rich features for writing JavaScript tests. Always start with Jest for writing tests for Lightning web components.

Jest tests are run at the command line or (with some configuration) within your IDE. Jest tests don't run in a browser or connect to an org, so they run fast. When run in "watch mode" they give you immediate feedback while you're coding. Jest tests work only with Lightning web components.

[Install Jest for Testing Lightning Web Components](#)

Install Jest and a few dependencies to use Jest with Lightning web components.

[Write Jest Tests for Lightning Web Components](#)

Write your component tests in local JavaScript files. Commit them to version control along with the component itself. Jest tests aren't saved to Salesforce.


[Run Jest Tests for Lightning Web Components](#)

Run your unit tests frequently during component development.

Install Jest for Testing Lightning Web Components

Install Jest and a few dependencies to use Jest with Lightning web components.

Testing with Jest has a few requirements beyond those for Salesforce DX and Lightning web components. Installation instructions vary by operating system. The following instructions are intended for use on platforms that offer a command line shell, such as Mac OS or Linux. The specific installation steps might be different for your operating system or version. However, the essential components remain the same.

 **Note:** We recognize that what follows is a bit complex and fiddly. It certainly might seem daunting to folks used to working exclusively with Salesforce tools. Relax, take it slow, and when necessary compare your progress to the [Lightning Web Components Samples](#) repo. We hope to integrate these industry standard tools more cleanly into Salesforce DX as Lightning Web Components progresses to General Availability, and beyond.

Prerequisites

The following components must be installed and working on your system before installing Jest for testing Lightning Web components.

- Salesforce DX — See [Get Started with Salesforce DX](#).
- Node.js — See <https://nodejs.org/>. There are two current releases of Node.js. We recommend using the "LTS" (Long Term Support) version, rather than the "Current" version.

- NPM — See <https://www.npmjs.com>.

You might already have Node installed on your system. To check, run the following command in your command shell.

```
node --version
```

You should get a result that resembles `v8.11.2`. If you get no output, or an earlier version, you'll need to install the latest LTS version.

To install the current release (as of this writing) of Node LTS using NVM, run the following command in your command shell.

```
nvm install 8.11.2
```

 **Note:** If you don't have NVM installed, follow the [NVM installation instructions on GitHub](#).

Install Jest and Its Dependencies into Your Project

Install tools and configuration details required to use Jest with Lightning web components with the following steps.

First, if it doesn't exist, create a `.npmrc` file for your project, and add the Lightning web components registry to it. The file belongs at the top level of your project, and should contain the following.

```
registry=https://npm.lwcjs.org
```

Alternatively, you can copy the `.npmrc` from the Lightning Web Components Samples repo.

Next, run the following commands from the top directory of your project.

```
npm install
npm install lts-jest --save-dev
```

Next, add an entry to identify your test script in the `scripts` block of your project's `package.json` file. The line should include a name and command. Your entry should look something like the following. (The added entry is emphasized.) We've used the name `test:unit` to run the `lts-jest test` command.

```
{
  ...
  "scripts": {
    ...
    "test:unit": "lts-jest test",
    ...
  },
  ...
}
```

Finally, you should now be able to run your project's Jest tests with the following command (replace `test:unit` with your entry's name).

```
npm run test:unit
```

The final result should be your first successful test run.



```


~/Development/src/sfdx-lwc-samples — -bash — 80x24
[sfdx-lwc-samples (master)]$ npm run test:unit

> sfdx-lwc-samples@0.0.1 test:unit /Users/malderete/Development/src/sfdx-lwc-samples
> lts-jest test

PASS force-app/main/default/lightningcomponents/hello_world/__tests__/hello_world-test.js
PASS force-app/main/default/lightningcomponents/todo/__tests__/todo-test.js

Test Suites: 2 passed, 2 total
Tests: 11 passed, 11 total
Snapshots: 0 total
Time: 2.206s, estimated 4s
Ran all test suites.
[sfdx-lwc-samples (master)]$

```

 **Note:** You must run these commands once inside each of your Lightning web components Salesforce DX projects. We recommend you first run them on a clean version of the Lightning Web Components Samples repo.

SEE ALSO:

[Explore Lightning Web Components Sample Code](#)

Write Jest Tests for Lightning Web Components

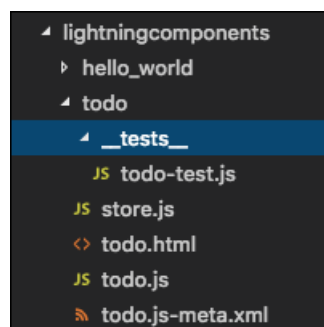
Write your component tests in local JavaScript files. Commit them to version control along with the component itself. Jest tests aren't saved to Salesforce.

Jest tests are written, saved, and run differently than Jasmine or Mocha tests written for the Lightning Testing Service. Jest tests are local only, and are saved and run independently of Salesforce.

Component Folder Structure

Create a folder named `__tests__` at the top level of your component's bundle directory. Save all tests for this component inside the `__tests__` folder. Share tests with other team members or systems by committing the `__tests__` folder to version control.

Example component bundle with `__tests__` folder



Use the project `.forceignore` file to ensure that the `__tests__` folder and its contents are ignored by Salesforce DX commands that push, pull, or transform code and metadata, and are never saved to Salesforce. Add the following glob pattern to the `.forceignore` file for each of your projects.

```
**/__tests__/**
```

Test File Naming Conventions

Jest runs all JavaScript files in the `__tests__` directory. Test files must have names that end in `.js`, and we recommend that tests end in `.test.js` or `-test.js`. You can have a single test file with all of your component tests, or you can have multiple files to organize and group related tests. Test files can be placed in sub folders.

Writing a Basic Test

To become an accomplished tester, you'll need to learn how to use Jest. In particular, you'll need to learn the syntax of the many [matchers provided with Jest](#). We won't cover general Jest usage here, because [the Jest documentation](#) is excellent. We'll instead focus on the specifics of using Jest with Lightning web components.

Jest tests for a Lightning web component should test the behavior of the component in isolation, with minimal dependencies on external components or services. Tests do this by creating an instance of the component, exercising its public API, and then either evaluating the DOM output of the component, or again using the public API to evaluate internal state.

Here's an example of a simple test.

```
// createElement is what we'll use to create our component under test
import { createElement } from 'engine';

// Import module under test by convention <namespace>-<moduleName>
import LwcHelloWorld from 'c-hello_world';

describe('hello_world', () => {
  it('displays expected header text', () => {
    const expectedText = 'Hello, Lightning web components';

    // Create an instance of the component under test
    const element = createElement('c-hello_world', { is: LwcHelloWorld });

    // Add component to DOM (jsdom) so it renders, goes through creation lifecycle
    document.body.appendChild(element);

    // Find the header element we want to inspect
    const header = element.querySelector('h1');

    // Compare the found text with the expected value
    expect(header.textContent).toBe(expectedText);

    // Available "expect" APIs can be found here:
    // https://facebook.github.io/jest/docs/en/expect.html
  });
});
```

The test creates an instance of the component we want to test, and then adds it to the test's version of `document`. The next step is to use a standard DOM query method to search the DOM for the element that was added. Finally, the `expect` statement is an assertion of two success conditions: (1) that the expected element was indeed found, and (2) that the found element has an expected style class associated with it.

Since there's no browser actually running, `document` is provided by `jsdom`, a part of Node.js. `jsdom` behaves much like `document` does inside a real browser, and your tests can, for the most part, treat it the same. However, each test file shares a single instance of

`jsdom`, and changes aren't reset between tests inside the file. Therefore it's a best practice to clean up between tests, so that each test's output doesn't affect any other test. To do this automatically, add code like the following to your test file.

```
afterEach(() => {
  // jsdom instance shared across test cases in a single file
  while (document.body.firstChild) {
    document.body.removeChild(document.body.firstChild);
  }
});
```

Testing Asynchronous DOM Updates

When the state of a Lightning web component changes, the DOM updates asynchronously. To ensure your test waits for updates to complete before evaluating the result, return a resolved Promise and chain the rest of your test code to it. Jest waits for the Promise chain to complete before ending the test. Jest fails the test if the Promise ends in the rejected state.

For example:

```
test('element does not have slds-icon class when bare', () => {
  const element = createElement('one-primitive-icon', { is: PrimitiveIcon });
  element.variant = "bare";

  // Use a promise to wait for asynchronous changes to the DOM
  return Promise.resolve().then(() => {
    expect(element.classList).not.toContain('slds-icon');
  });
});
```

SEE ALSO:

[Jest: Expect Reference](#)

Run Jest Tests for Lightning Web Components

Run your unit tests frequently during component development.

Run Tests on the Command Line

To run all tests for your project, run the command you added to the `scripts` block of your project's `package.json` file. See [Install Jest for Testing Lightning Web Components](#) for those instructions. The line will be similar to the following.

```
npm run test:unit
```

Run Tests Continuously During Development

To run all tests for a single component every time you save changes, change directories to the component directory, and run the command you added to the `scripts` block of your project's `package.json` file with the `--watch` parameter.

```
npm run test:unit -- --watch
```

Jest watches all component files for updates, and runs all relevant tests every time it detects a change.

CHAPTER 10 Debugging

In this chapter ...

- [Enable Debug Mode](#)

There are a few basic tools and techniques that can help you to debug applications.

Use Chrome DevTools to debug your client-side code.

- To open DevTools on Windows and Linux, press Control-Shift-I in your Google Chrome browser. On Mac, press Option-Command-I.
- To quickly find which line of code is failing, enable the **Pause on all exceptions** option before running your code.

To learn more about debugging JavaScript on Google Chrome, refer to the [Google Chrome's DevTools](#) website.


Enable Debug Mode

Enable debug mode to make it easier to debug JavaScript code in Lightning web components. Only enable debug mode for users who are actively debugging JavaScript. Salesforce is slower for users who have debug mode enabled.

The Lightning Component framework executes in one of two modes: production and debug.


Production Mode

By default, the framework runs in production mode. This mode is optimized for performance. Framework code is optimized and “minified” to reduce the size of the JavaScript code. As a result of this process, the JavaScript code served to the browser is obfuscated.

 **Important:** Minification is a performance optimization, not intellectual property protection. Code that’s minified is hard to read, but it’s not encrypted or otherwise prevented from being viewed.

Debug Mode

When you enable debug mode, framework JavaScript code isn’t minified and is easier to read and debug. Debug mode also adds more detailed output for some warnings and errors. As with production mode, custom component code is not optimized or minified.

 **Important:** Debug mode has a significant performance impact. Salesforce is slower for any user who has debug mode enabled. For this reason, we recommend using it only when actively debugging JavaScript code, and only for users involved in debugging activity. Don’t leave debug mode on permanently. Users who have debug mode enabled see a banner notification once a week while it’s enabled.

To enable debug mode for users in your org:

1. From Setup, enter *Debug Mode* in the *Quick Find* box, then select **Debug Mode Users**.
Users with debug mode enabled have a check mark in the Debug Mode column.
2. In the user list, locate any users who need debug mode enabled. If necessary, use the standard list view controls to filter your org’s users.
3. Enable the selection checkbox next to users for whom you want to enable debug mode.
4. Click **Enable**.

To disable debug mode for a user, follow the preceding steps and click **Disable** instead of **Enable**.

CHAPTER 11 Lightning Web Components Reference

In this chapter ...

- [Components](#)
- [HTML Template Directives](#)
- [Lifecycle Hooks](#)
- [lightning-ui-api* Wire Adapters and Functions](#)
- [Validations for Lightning Web Component Code](#)

The Lightning Web Components Developer Guide contains reference documentation for the Lightning Web Components development model, such as HTML template directives, lifecycle hooks, wire service adapters, and JavaScript APIs.

Components

To look up a component's attributes, and to explore its look and feel, use the Component Library.



Important: During the pilot release, not all base Lightning components are available as Lightning web components. If you try to use one that isn't available, you'll get a save error. We'll add more base Lightning web components as we move towards GA.

To look up a component's attributes, and to explore its look and feel, use the Component Library.

- <https://<myCustomSalesforceDomain>.lightning.force.com/componentReference/suite.app>
- <https://developer.salesforce.com/docs/component-library>

During the pilot release, Lightning web components aren't described in the Component Library, but their Aura equivalents are. Lightning web components use a slightly different naming convention, but they have the same attributes. To find the documentation for a Lightning web component, convert its name to the Aura naming convention.

For example, to find documentation for the `lightning-button-icon` Lightning web component, look for the Aura `lightning:buttonIcon` component.

- **lightning-accordion**
- **lightning-accordion-section**
- lightning-avatar
- lightning-badge
- lightning-breadcrumb
- lightning-breadcrumbs
- lightning-button: Doesn't support `body` because the Lightning web component doesn't have a slot.
- lightning-button-icon
- lightning-button-icon-stateful
- **lightning-button-menu**
- lightning-button-stateful
- **lightning-card**
- lightning-carousel
- lightning-checkbox-group
- lightning-click-to-dial
- lightning-combobox
- lightning-datatable
- lightning-dual-listbox
- lightning-dynamic-icon
- lightning-file-upload
- **lightning-flexipage-service**—See [Make Your Component Width-Aware with lightning-flexipage-service](#).
- lightning-formatted-address
- lightning-formatted-date-time
- lightning-formatted-email
- lightning-formatted-location
- lightning-formatted-name
- lightning-formatted-number

- `lightning-formatted-phone`
- `lightning-formatted-rich-text`
- `lightning-formatted-text`
- `lightning-formatted-time`
- `lightning-formatted-url`
- `lightning-helptext`
- `lightning-icon`
- `lightning-input`
- `lightning-input-address`
- `lightning-input-field`
- `lightning-input-location`
- `lightning-input-name`
- `lightning-input-rich-text`: To add buttons to the toolbar of the rich text editor, use the `custom-buttons` attribute instead of the `Aura body` facet. Pass the buttons as an array of objects.
- **`lightning-layout`**
- **`lightning-layout-item`**
- **`lightning-menu-item`**
- `lightning-navigation`: See [Navigate to Pages in Lightning Experience and Salesforce](#).
- **`lightning-notifications-library`**
- `lightning-output-field`
- `lightning-overlay-library`
- `lightning-pill`: Use a slot instead of the `Aura media` attribute. The Lightning web component also has a `variant` attribute, which changes the pill type to be a clickable link (`'link'`) or not (`'plain'`).
- `lightning-pill-container`
- `lightning-progress-indicator`
- `lightning-radio-group`
- `lightning-record-edit-form`
- `lightning-record-form`
- `lightning-record-view-form`
- `lightning-relative-date-time`
- `lightning-slider`
- `lightning-spinner`
- `lightning-textarea`
- `lightning-tree`
- `lightning-tree-grid`
- `lightning-vertical-navigation-item`
- `lightning-vertical-navigation-item-badge`
- `lightning-vertical-navigation-item-icon`

HTML Template Directives

A directive is a special attribute that adds dynamic behavior to the HTML template.

if:true|false={expression}

Use this directive to conditionally render DOM elements in a template.

The **expression** can be a JavaScript identifier (for example, `person`) or dot notation that accesses a property from an object (`person.firstName`). The Lightning Web Components model doesn't allow computed expressions (`person[2].name['John']`). To compute the value of **expression**, use a [getter](#) in the JavaScript class.

In this example, the `if:true` directive displays the `<p>` element based on the value of `show` from the component's instance. In this example, the component shows the text.

```
<template>
  <p if:true={show}>This content is visible.</p>
</template>
```

```
import { Element } from "engine";
export default class MyComponent extends Element {
  show = true;
}
```

for:each={array}

Use this directive to iterate over an array.

for:item="currentItem"

Use this directive to access the current item.

The `currentItem` placeholder is a new identifier that the model injects into the current scope.

for:index="index"

Use this directive to access the current item's zero-based index.

The `index` placeholder is a new identifier that the model injects into the current scope.

```
<template>
  <div class="my-list" for:each={parks} for:item="item" for:index="index">
    <p>{index} - {item.name}</p>
  </div>
</template>
```

```
import { Element } from "engine";
export default class MyComponent extends Element {
  parks = [
    { name: 'Lassen Volcanic National Park' },
    { name: 'Channel Islands National Park' },
    { name: 'Pinnacles National Park' },
  ];
}
```

key={uniqueId}

Use this directive to improve rendering performance by assigning a unique identifier to each item in a list. The `key` must be a string or a number, it can't be an object. The Lightning Web Components model uses the keys to determine which items have changed.

```
<template>
  <ul for:each={parks} for:item="park" for:index="index">
```

```
<li key={park.id}>{index}-{park.name}</li>
</ul>
</template>
```

iterator: iteratorName={array}

Use this directive to apply special behavior to the first or last item in an array.

You can access these properties on the iterator:

- `value`—The value of the item in the list.
- `index`—The index of the item in the list.
- `first`—A boolean value indicating whether this item is the first item in the list.
- `last`—A boolean value indicating whether this item is the last item in the list.

```
<template>
  <template class="my-list" iterator:it={items}>
    <h1 if:true={it.first}>{it.value}</h1>
    <p if:false={it.first}>{it.value}</p>
  </template>
</template>
```

```
import { Element } from "engine";

export default class App extends Element {
  items = [1, 2, 3, 5];
}
```

SEE ALSO:

[Render Lists](#)

Lifecycle Hooks

A lifecycle hook is a callback method triggered at a specific phase of a component instance's lifecycle.

constructor()

Called when the component is created. This hook flows from parent to child. You can't access child elements in the component body because they don't exist yet. Properties are not passed yet, either. Properties are assigned to the component after construction and before the `connectedCallback()` hook. You can access the host element with `this.root`.

connectedCallback()

Called when the element is inserted into a document. This hook flows from parent to child. You can't access child elements in the component body because they don't exist yet. You can access the host element with `this.root`.

disconnectedCallback()

Called when the element is removed from a document. This hook flows from parent to child.

render()

For complex tasks like conditionally rendering a template or importing a custom one, use `render()` to override standard rendering functionality. This function gets invoked after `connectedCallback()` and must return a valid HTML template.

renderedCallback()

Called after every render of the component. This lifecycle hook is specific to Lightning Web Components, it isn't from the HTML custom elements specification. This hook flows from child to parent.

If you use `renderedCallback()` to perform a one-time operation, you must track it manually (using an `initialRender` private property, for example). If you perform changes to reactive attributes, guard them or they can trigger wasteful rerenders or an infinite rendering loop.

`errorCallback(error, stack)`

Called when a descendant component throws an error in one of its lifecycle hooks. The `error` argument is a JavaScript native error object, and the `stack` argument is a string. This lifecycle hook is specific to Lightning Web Components, it isn't from the HTML custom elements specification.

Implement this hook to create an error boundary component that captures errors in all the descendent components in its tree. The error boundary component can log stack information and render an alternative view to tell users what happened and what to do next. The method works like a JavaScript `catch { }` block for components that throw errors in their lifecycle hooks. It's important to note that an error boundary component catches errors only from its children, and not from itself.

SEE ALSO:

[Component Lifecycle Hooks](#)

lightning-ui-api* Wire Adapters and Functions

Wire adapters and JavaScript functions in these modules are built on top of Lightning Data Service (LDS) and User Interface API. Use these wire adapters and functions to work with Salesforce data and metadata.

For information about how LDS and User Interface API work, see [User Interface API Wire Adapters and JavaScript Functions](#).

[Supported Salesforce Objects](#)

Lightning web components access Salesforce data and metadata via User Interface API. User Interface API supports all custom objects and many standard objects.

[lightning-ui-api-list-ui](#)

Get the records and metadata for a list view.

[lightning-ui-api-lookups](#)

When a user edits a lookup field, use this resource to search for and display suggestions for a specified object. You can search for most recently used matches, for matching names, or for any match in a searchable field. You can also specify lookup filter bindings for dependent lookups.

[lightning-ui-api-object-info](#)

Get object metadata, and get picklist values.

[lightning-ui-api-record](#)

This module includes wire adapters to record data and get default values to create records. It also includes JavaScript APIs to create and update records.

[lightning-ui-api-record-ui](#)

Get layout information, metadata, and data to build UI for a single record or for a collection of records.

SEE ALSO:

[User Interface API Wire Adapters and JavaScript Functions](#)

Supported Salesforce Objects

Lightning web components access Salesforce data and metadata via User Interface API. User Interface API supports all custom objects and many standard objects.

Lightning web components can access Salesforce data and metadata from all custom objects and from all the standard objects in this list.

- Account
- AccountTeamMember
- Asset
- AssetRelationship
- AssignedResource
- AttachedContentNote
- BusinessAccount
- Campaign
- CampaignMember
- Case
- Contact
- ContentDocument
- ContentNote
- ContentVersion
- ContentWorkspace
- Contract
- ContractContactRole
- ContractLineItem
- Custom Object
- Entitlement
- EnvironmentHubMember
- Lead
- LicensingRequest
- MaintenanceAsset
- MaintenancePlan
- MarketingAction
- MarketingResource
- Note
- OperatingHours
- Opportunity
- OpportunityLineItem
- OpportunityTeamMember
- Order
- OrderItem
- PersonAccount

- Pricebook2
- PricebookEntry
- Product2
- Quote
- QuoteDocument
- QuoteLineItem
- ResourceAbsence
- ResourcePreference
- ServiceAppointment
- ServiceContract
- ServiceCrew
- ServiceCrewMember
- ServiceResource
- ServiceResourceCapacity
- ServiceResourceSkill
- ServiceTerritory
- ServiceTerritoryLocation
- ServiceTerritoryMember
- Shipment
- SkillRequirement
- SocialPost
- Tenant
- TimeSheet
- TimeSheetEntry
- TimeSlot
- UsageEntitlement
- UsageEntitlementPeriod
- User
- WorkOrder
- WorkOrderLineItem
- WorkType

lightning-ui-api-list-ui

Get the records and metadata for a list view.

[getListUi](#)

Get the records and metadata for a list view.

getListUi

Get the records and metadata for a list view.

Get list view records and metadata for a list view by API name

Syntax

```
import { Element, wire } from 'engine';
import { getListUi } from 'lightning-ui-api-list-ui';
import Account from '@salesforce/schema/Account';

export default class Example extends Element {
  @wire(getListUi, { objectApiName: Account, listViewApiName: 'AllAccounts' })
  accountListUiByApiName;
}
```

- `objectApiName`—(Required) The API name of a [supported object](#).
- `listViewApiName`—(Required) The API name of a list view, such as AllAccounts.

You can also pass the parameters listed in this [Request Parameters table](#).

Returns

- `value.data`—[List UI](#)
- `value.error`—[Error Message](#)

Get list view records and metadata for a list view by ID

Syntax

```
import { Element, wire } from 'engine';
import { getListUi } from 'lightning-ui-api-list-ui';
export default class Example extends Element {
  @wire(getListUi, { listViewId: '00BT0000001TONQMA4' })
  accountListUiById;
}
```

- `listViewId`—(Required) The ID of a list view.

You can also pass the parameters listed in this [Request Parameters table](#).

Returns

- `value.data`—[List UI](#)
- `value.error`—[Error Message](#)

Get list view records and metadata for an MRU list view by object

Syntax

```
import { Element, wire } from 'engine';
import { getListUi, MRU } from 'lightning-ui-api-list-ui';
import Account from '@salesforce/schema/Account';

export default class Example extends Element {
  @wire(getListUi, { objectApiName: Account, listViewApiName: MRU })
  accountMruListUi;
}
```

- `objectApiName`—(Required) The API name of a [supported object](#).

- `listViewApiName`—(Required) The API name of the MRU list view.

You can also pass the parameters listed in this [Request Parameters table](#).

Returns

- `value.data`—[MRU List Record Collection](#)
- `value.error`—[Error Message](#)

Get list views for an object

Syntax

```
import { Element, wire } from 'engine';
import { getListUi } from 'lightning-ui-api-list-ui';
import Account from '@salesforce/schema/Account';

export default class Example extends Element {
  @wire(getListUi, { objectApiName: Account })
  accountListOfListViews;
}
```

- `objectApiName`—(Required) The API name of a [supported object](#).

You can also pass the parameters listed in this [Request Parameters table](#).

Returns

- `value.data`—[List View Summary Collection](#)
- `value.error`—[Error Message](#)

SEE ALSO:

[Use the Wire Service to Get Data](#)

lightning-ui-api-lookups

When a user edits a lookup field, use this resource to search for and display suggestions for a specified object. You can search for most recently used matches, for matching names, or for any match in a searchable field. You can also specify lookup filter bindings for dependent lookups.

[getLookupRecords](#)

Get lookup field suggestions for a specified object.

getLookupRecords

Get lookup field suggestions for a specified object.

Syntax

```
import { Element, wire } from 'engine';
import { getLookupRecords } from 'lightning-ui-api-lookups';
import Account from '@salesforce/schema/Account';
import Opportunity_AccountId from '@salesforce/schema/Opportunity.AccountId';
export default class Example extends Element {
  @wire(getLookupRecords, { fieldApiName: Opportunity_AccountId,
```

```
        targetApiName: Account })
    accountFieldSuggestions;
}
```

- `fieldName`—(Required) The API name of a lookup field.
- `targetApiName`—(Required) The API name of the target (lookup) object. The object must be a [supported object](#).

You can also pass the parameters listed in this [Request Parameters table](#).

Returns

- `value.data`—[Record Collection](#)
- `value.error`—[Error Message](#)

SEE ALSO:

[Use the Wire Service to Get Data](#)

lightning-ui-api-object-info

Get object metadata, and get picklist values.

[getObjectInfo](#)

Get metadata about a specific object. The response includes metadata describing fields, child relationships, record type, and theme.

[getPicklistValues](#)

Get the picklist values for a specified field.

[getPicklistValuesByRecordType](#)

Get the values for every picklist of a specified record type.

getObjectInfo

Get metadata about a specific object. The response includes metadata describing fields, child relationships, record type, and theme.

Syntax

```
import { Element, wire } from 'engine';
import { getObjectInfo } from 'lightning-ui-api-object-info';
import Account from '@salesforce/schema/Account';

export default class Example extends Element {
  @wire(getObjectInfo, { objectApiName: Account })
  accountObjectInfo;
}
```

- `objectApiName`—(Required) A [supported object](#).

Returns

- `value.data`—[Object Info](#)

- `value.error`—[Error Message](#)

SEE ALSO:

[Use the Wire Service to Get Data](#)

[Use the Wire Service to Get Data](#)

getPicklistValues

Get the picklist values for a specified field.

Syntax

```
import { Element, wire } from 'engine';
import { getPicklistValues } from 'lightning-ui-api-object-info';
import Industry from '@salesforce/schema/Account.Industry';

export default class Example extends Element {
  @wire(getPicklistValues, { recordTypeId: '012000000000000AAA', fieldApiName: Industry
  })
  industryFieldPicklistValues;
}
```

- `recordTypeId`—(Required) The ID of the record type.
- `fieldApiName`—(Required) The API name of the picklist field on a [supported object](#).

Returns

- `value.data`—[Picklist Values](#)
- `value.error`—[Error Message](#)

SEE ALSO:

[Use the Wire Service to Get Data](#)

[Use the Wire Service to Get Data](#)

getPicklistValuesByRecordType

Get the values for every picklist of a specified record type.

Syntax

```
import { Element, wire } from 'engine';
import { getPicklistValuesByRecordType } from 'lightning-ui-api-object-info';
import Account from '@salesforce/schema/Account';

export default class Example extends Element {
  @wire(getPicklistValuesByRecordType, { objectApiName: Account, recordTypeId:
  '012000000000000AAA' })
  accountPicklistValues
}
```

- `objectApiName`—(Required) The API name of a [supported object](#).
- `recordTypeId`—(Required) The ID of the record type.

Returns

- `value.data`—[Picklist Values Collection](#)
- `value.error`—[Error Message](#)

SEE ALSO:

[Use the Wire Service to Get Data](#)

lightning-ui-api-record

This module includes wire adapters to record data and get default values to create records. It also includes JavaScript APIs to create and update records.

[createRecord\(recordInput\)](#)

Creates a record.

[createRecordInputFilteredByEditedFields\(recordInput, originalRecord\)](#)

Creates a RecordInput object with a list of fields that have been edited from their original values to pass in a call to `updateRecord(recordInput)`.

[deleteRecord\(recordId\)](#)

Deletes a record.

[generateRecordInputForCreate\(record\)](#)

Creates a record object filtered to contain updatable fields. Pass this object in a call to `createRecord(recordInput)`.

[generateRecordInputForUpdate\(record\)](#)

Creates a record object filtered to contain updatable fields. Pass this object in a call to `updateRecord(recordInput)`.

[getRecord](#)

Get a record's data.

[getRecordCreateDefaults](#)

Get the default layout information and object information for creating a record.

[getRecordInput\(\)](#)

Get an object that represents a new or updated record to be saved on the server.

[updateRecord\(recordInput\)](#)

Updates a record.

createRecord(recordInput)

Creates a record.

Syntax

```
import { createRecord } from 'lightning-ui-api-record';
createRecord(recordInput: Record): Promise<Record>
```

- `recordInput`—(Required) A [RecordInput](#) object used to create the record. To create a RecordInput object, use `generateRecordInputForCreate(record)`.

Returns

A Promise object that resolves with the created record. The record contains data for the fields in the record layout.

createRecordInputFilteredByEditedFields (recordInput, originalRecord)

Creates a RecordInput object with a list of fields that have been edited from their original values to pass in a call to `updateRecord(recordInput)`.

Syntax

```
import { createRecordInputFilteredByEditedFields } from 'lightning-ui-api-record';
createRecordInputFilteredByEditedFields(recordInput: Record, originalRecord: Record):
Record
```

- `recordInput`—(Required) A [RecordInput](#) object to filter.
- `originalRecord`—(Required) A Record object that contains the original field values. The structure of the record object is:

```
{
  'apiName': 'ObjectApiName',
  'fields': {
    'Name': {
      'value': 'SomeValue'
    },
    'Description': {
      'value': 'SomeValue'
    }
  }
}
```

Returns

An object with a list of fields that have been edited from their original values (excluding `Id` which is always included).

deleteRecord (recordId)

Deletes a record.

Syntax

```
import { deleteRecord } from 'lightning-ui-api-record';
deleteRecord(recordId: Record): Void
```

- `recordId`—(Required) The ID of the record to delete.

generateRecordInputForCreate (record)

Creates a record object filtered to contain updatable fields. Pass this object in a call to `createRecord(recordInput)`.

Syntax

```
import { generateRecordInputForCreate } from 'lightning-ui-api-record';
generateRecordInputForCreate(record: Record, objectInfo: ObjectInfo): Record
```

- **record**—(Required) A Record object that contains source data. The structure of the Record object is:

```
{
  'apiName': 'ObjectApiName',
  'fields': {
    'Name': {
      'value': 'SomeValue'
    },
    'Description': {
      'value': 'SomeValue'
    }
  }
}
```

To get data to build the Record object, use the `getRecordCreateDefaults` wire adapter.

- **objectInfo**—(Optional) The ObjectInfo corresponding to the `apiName` on the record. If provided, only fields that are `updatable=true` (excluding `Id`) are included in the response.

Returns

An object with its data populated from the given record. Returns all fields whose values are not nested records.

generateRecordInputForUpdate (record)

Creates a record object filtered to contain updatable fields. Pass this object in a call to `updateRecord (recordInput)`.

Syntax

```
import { generateRecordInputForUpdate } from 'lightning-ui-api-record';
generateRecordInputForUpdate(record: Record, objectInfo: ObjectInfo): Record
```

- **record**—(Required) A Record object that contains source data. The structure of the Record object is:

```
{
  'apiName': 'ObjectApiName',
  'fields': {
    'Name': {
      'value': 'SomeValue'
    },
    'Description': {
      'value': 'SomeValue'
    }
  }
}
```

- **objectInfo**—(Optional) The ObjectInfo corresponding to the `apiName` on the record. If provided, only fields that are `updatable=true` (excluding `Id`) are included in the response.

Returns

An object with its data populated from the given record. Returns all fields whose values are not nested records.

getRecord

Get a record's data.

Syntax

```
import { Element, wire } from 'engine';
import { getRecord } from 'lightning-ui-api-record';
export default class Example extends Element {
  @wire(getRecord, { recordId: '001456789012345678', fields: [NameField] })
  record;
}
```

- **recordId**—(Required) The ID of a record from a [supported object](#).
- **fields**—(Required) An array of fields to return. If the context user doesn't have access to a field, an error is returned. If you're not sure whether the context user has access to a field and you don't want the request to fail if they don't, use the **optionalFields** parameter.
- **optionalFields**— (Optional) An array of optional field names. If a field is accessible to the context user, it's included in the response. If a field isn't accessible to the context user, it isn't included in the response, but it doesn't cause an error.

For both the **fields** and **optionalFields** parameters, specify field names in the format **ObjectName.FieldName** or **ObjectName.JunctionIdListName**.

 **Important:** The pilot release doesn't support **childRelationships**.

Returns

- **value.data**—[Record](#)
- **value.error**—[Error Message](#)

SEE ALSO:

[Use the Wire Service to Get Data](#)

[Use the Wire Service to Get Data](#)

getRecordCreateDefaults

Get the default layout information and object information for creating a record.

Syntax

```
import { Element, wire } from 'engine';
import { getRecordCreateDefaults } from 'lightning-ui-api-record';
import Account from '@salesforce/schema/Account';
export default class Example extends Element {
  @wire(getRecordCreateDefaults, { objectApiName: Account })
  accountMetadata;
}
```

- **apiName**— (Required) A [supported object](#).
- **formFactor**— (Optional) The layout display size for the record. An array containing any of these values:
 - **Large**—(Default) Use this value to get a layout for desktop display size.
 - **Medium**—Use this value to get a layout for tablet display size.
 - **Small**—Use this value to get a layout for phone display size.
- **recordTypeId**— (Optional) The ID of the record type (RecordType object) for the new record. If not provided, the default record type is used.

- `optionalFields`— (Optional) An array of fields to return along with the default fields. If an optional field is accessible to the context user, it's included in the response. If it isn't accessible to the context user, it isn't included in the response, but it doesn't cause an error.

Returns

- `value.data`—[Record Defaults](#)
- `value.error`—[Error Message](#)

Usage

To create UI that lets a user create a record, first get information about which fields are required. This adapter's response contains the default field values for a new record of the object type specified in `{ apiName }`. It also contains object metadata and the corresponding layout for Create mode. In the Salesforce user interface, an admin with "Customize Application" permission can mark a field as required in a layout. When you're building UI, to determine which fields to mark as required in a layout for create and update, use the `ObjectInfo.fields[fieldName].required` property.



Example: https://github.com/forcedotcom/ebikes-lwc/tree/master/force-app/main/default/lightningcomponents/order_builder

SEE ALSO:

[Use the Wire Service to Get Data](#)

[Use the Wire Service to Get Data](#)

getRecordInput()

Get an object that represents a new or updated record to be saved on the server.

Syntax

```
import { getRecordInput } from 'lightning-ui-api-record';
getRecordInput(): Record
```

Returns

Returns an object that represents a new or updated record to be saved on the server. See [Record Input](#).

updateRecord(recordInput)

Updates a record.

Syntax

```
import { updateRecord } from 'lightning-ui-api-record';
updateRecord(recordInput: Record, clientOptions: Object): Promise<Record>
```

- `recordInput`— (Required) A [RecordInput](#) object used to update the record. To create the object, call [generateRecordInputForUpdate\(record\)](#).
- `clientOptions`— (Optional) To check for conflicts before you update a record, pass `const clientOptions = { 'ifUnmodifiedSince' : lastModifiedDate }`. Get `lastModifiedDate` via a wire service adapter that returns a record object: `const lastModifiedDate = record.fields.LastModifiedDate.value;`

Returns

A Promise object that resolves with the updated record. The record contains data for the fields in the record layout.

lightning-ui-api-record-ui

Get layout information, metadata, and data to build UI for a single record or for a collection of records.

[getRecordUi](#)

Get layout information, metadata, and data to build UI for a single record or for a collection of records.

getRecordUi

Get layout information, metadata, and data to build UI for a single record or for a collection of records.

Syntax

```
import { Element, wire } from 'engine';
import { getRecordUi } from 'lightning-ui-api-record-ui';
export default class Example extends Element {
  @wire(getRecordUi, { recordIds: ['001456789012345678'], layoutTypes: ['Full'], modes:
    ['View'] })
    accountRecordUi;
}
```

- **recordIds**—(Required) An array of records from [supported objects](#).
- **layoutTypes**—(Required) Layout types for the record. An array containing any of these values:
 - **Compact**—Use this value to get a layout that contains a record's key fields.
 - **Full**—(Default) Use this value to get a full layout.
- **modes**—(Required) The access mode for the record. This value determines which fields to get from a layout. Layouts have different fields for create, edit, and view modes. For example, formula fields are rendered in view mode, but not in create mode because they're calculated at run time, like formulas in a spreadsheet. An array containing any of these values:
 - **Create**—Use this mode if you intent to build UI that lets a user create a record. This mode is used by the `record-create-defaults` wire adapter.
 - **Edit**—Use this mode if you intent to build UI that lets a user edit a record. This mode is used by the `record-clone-defaults` wire adapter.
 - **View**—(Default) Use this mode if you intent to build UI that displays a record.
- **optionalFields**—(Optional) An array of optional field names. If a field is accessible to the context user, it's included in the response. If a field isn't accessible to the context user, it isn't included in the response, but it doesn't cause an error.

Returns

- **value.data**—[Record UI](#)
- **value.error**—[Error Message](#)



Example:

https://github.com/forcedotcom/sfdx-lwc-samples/tree/master/force-app/main/default/lightningcomponents/lds_record_ui

SEE ALSO:

[Use the Wire Service to Get Data](#)

[Use the Wire Service to Get Data](#)

Validations for Lightning Web Component Code

Validate your Lightning Web component code to ensure compatibility with Lightning component APIs and best practices, and to avoid anti-patterns.

There are two ways to install the recommended ESLint configuration.

- Install the [LWC Code Editor for Visual Studio Code](#), which automatically configures and activates the recommended ESLint configuration.
- Install the ESLint configuration from the command line. See the [eslint-plugin-lwc repository](#) for instructions.

We've created some rules specifically for Lightning web component development to minimize common mistakes. They are listed here with information to help you resolve any issues you encounter.

In addition to the rules we've created, other rules are included from ESLint basic rules. Documentation for these rules is available on the ESLint project site. If you encounter an error or warning from a rule not described here, search for it on [the ESLint Rules](#) page.

Disallow Use of Aura Libraries

Aura libraries are not compatible with Lightning web components. Convert the logic within your Aura libraries to Lightning Web Component compatible modules and directly import the converted libraries.

Message: Do not use an aura library from LWC

Details

Import statements can't contain a colon, which is an indicator that you're importing an Aura library.

Incorrect:

```
import { foo } from 'ui:something';
```

Correct:

```
import { foo } from 'ui-something';
```

Disallow Use of \$A

Don't use \$A functions in Lightning web components. Use native DOM/BOM APIs instead.

Message: Do not use \$A in LWC code

Details

Ensure there are no references to \$A.

Incorrect:

```
$A.util.isArray([]);
```

Correct:

```
Array.isArray([]);
```

Restrict Use of Compatibility API `createComponent`

For backwards compatibility, the Lightning Web Components model allows access to some \$A APIs via an import of the `aura` module. However, use of these APIs is strongly discouraged because the \$A APIs will be deprecated in a future release.

Message: Do not use `'create Component'` from `'aura'`

Details

Don't import `create Component` from `aura`.

Incorrect:

```
import { create Component } from 'aura';
```

Restrict Use of Compatibility API `dispatchGlobalEvent`

For backwards compatibility, the Lightning Web Components model allows access to some \$A APIs via an import of the `aura` module. However, use of these APIs is strongly discouraged because the \$A APIs will be deprecated in a future release.

Message: Do not use `'dispatchGlobalEvent'` from `'aura'`

Details

Don't import `dispatchGlobalEvent` from `aura`.

Incorrect:

```
import { dispatchGlobalEvent } from 'aura';
```

Restrict Use of Compatibility API `executeGlobalController`

For backwards compatibility, the Lightning Web Components model allows access to some \$A APIs via an import of the `aura` module. However, use of these APIs is strongly discouraged because the \$A APIs will be deprecated in a future release.

Message: Do not use `'executeGlobalController'` from `'aura'`

Details

Don't import `executeGlobalController` from `aura`.

Incorrect:

```
import { executeGlobalController } from 'aura';
```

Restrict Import of Compatibility module `aura-instrumentation`

For backwards compatibility, the Lightning Web Components model allows access to some \$A APIs via an import of the `aura-instrumentation` module. However, use of these APIs is strongly discouraged because the \$A APIs will be deprecated in a future release.

Message: Do not `import` the entire `'aura-instrumentation'` module

Details

Don't import `aura-instrumentation`.

Incorrect:

```
import { time } from 'aura-instrumentation';
```

Restrict Import of Compatibility module `aura-storage`

For backwards compatibility, the Lightning Web Components model allows access to some \$A APIs via an import of the `aura-storage` module. However, use of these APIs is strongly discouraged because the \$A APIs will be deprecated in a future release.

Message: Do not `import` the entire `'aura-storage'` module

Details

Don't import `aura-storage`.

Incorrect:

```
import { getstorage } from 'aura-storage';
```

Restrict Use of Compatibility API `registerModule`

For backwards compatibility, the Lightning Web Components model allows access to some \$A APIs via an import of the `aura` module. However, use of these APIs is strongly discouraged because the \$A APIs will be deprecated in a future release.

Message: Do not use `'registerModule'` from `'aura'`

Details

Don't import `registerModule` from `aura`.

Incorrect:

```
import { registerModule } from 'aura';
```

Restrict Use of Compatibility API `sanitizeDOM`

For backwards compatibility, the Lightning Web Components model allows access to some \$A APIs via an import of the `aura` module. However, use of these APIs is strongly discouraged because the \$A APIs will be deprecated in a future release.

Message: Do not use `'sanitizeDOM'` from `'aura'`

Details

Don't import `sanitizeDOM` from `aura`.

Incorrect:

```
import { sanitizeDOM } from 'aura';
```

Restrict Use of Compatibility module `aura`

For backwards compatibility, the Lightning Web Components model allows access to some \$A APIs via an import of the `aura` module. However, use of these APIs is strongly discouraged because the \$A APIs will be deprecated in a future release.

Message: Do not `import` the entire `'aura'` module

Details

Don't import the entire `aura` module.

Incorrect:

```
import * as foo from 'aura';
```


Disallow Disabling ESLint Via Inline Comments

You have multiple ways to disable enforcement of an ESLint rule in source code. The `eslint-comments/no-use` rule restricts disabling rules, but does not disable rules for inline comments. We restrict the usage of inline comments to disable ESLint rules so all ESLint rules are properly enforced.

Message: `Inline rule disablement not allowed`

Details

Don't use inline comments to disable ESLint rules using `// eslint-disable-line` or `// eslint-disable-next-line`.

Incorrect:

```
alert(); // eslint-disable-line

// eslint-disable-next-line
alert();
```

Disallow Use of `innerHTML`

Use of `innerHTML` may allow malicious JavaScript to execute unintentionally. When you're setting plain text, use `Node.textContent`. When you're interacting with DOM nodes, use the native DOM APIs.

Message: Using `'innerHTML/outputHTML/insertAdjacentHTML'` is not allowed

Details

Don't use `innerHTML` in all its forms, including `innerHTML`, `outputHTML`, and `insertAdjacentHTML`.

Incorrect:

```
element.innerHTML = '<foo></foo>';
element.outerHTML = '<foo></foo>';
element.insertAdjacentHTML = '<foo></foo>';
```

Correct:

```
element.textContent = 'foo';
```

Restrict Use of `process.env` to Whitelisted Set

Don't use the global variable `process.env` in a Node app to access or modify the global state. Instead, use `process.env.NODE_ENV` to see the mode the app is running.

Message: `process` is not allowed in LWC code

Details

Disallow access to anything on `process` or `process.env` other than `process.env.NODE_ENV`.

Incorrect:

```
process.env;
process.nextTick();
global.process;
```

Correct:

```
process.env.NODE_ENV;
```

Disallow Use of `window.requestAnimationFrame`

You can't use `requestAnimationFrame` because it impacts performance.

Message: Using '`requestAnimationFrame`' is not allowed for performance reasons

Details

Disallow use of `requestAnimationFrame` or `window.requestAnimationFrame` other than `process.env.NODE_ENV`.

Incorrect:

```
window.requestAnimationFrame(callback);
```

Disallow Use of `window.setInterval`

You can't use `setInterval` because it impacts performance.

Message: Using '`setInterval`' is not allowed for performance reasons

Details

Disallow use of `setInterval` or `window.setInterval`.

Incorrect:

```
window.setInterval(func, 100);
```

Disallow Use of `window.setTimeout`

You can't use `setTimeout` because it impacts performance.

Message: Using '`setTimeout`' is not allowed for performance reasons

Details

Disallow use of `setTimeout` or `window.setTimeout`.

Incorrect:

```
window.setTimeout(func, 100);
```

CHAPTER 12 Glossary

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

A

App

Short for “application.” A collection of components such as tabs, reports, dashboards, and Visualforce pages that address a specific business need. Salesforce provides standard apps such as Sales and Service. You can customize the standard apps to match the way you work. In addition, you can package an app and upload it to the AppExchange along with related components such as custom fields, custom tabs, and custom objects. Then, you can make the app available to other Salesforce users from the AppExchange.

B

No Glossary items for this entry.

C

Client App

An app that runs outside the Salesforce user interface and uses only the Lightning Platform API or Bulk API. It typically runs on a desktop or mobile device. These apps treat the platform as a data source, using the development model of whatever tool and platform for which they are designed.

Code Coverage

A way to identify which lines of code are exercised by a set of unit tests, and which are not. This helps you identify sections of code that are completely untested and therefore at greatest risk of containing a bug or introducing a regression in the future.

Container

A container contains other components but itself is contained within the owner component. A container is less powerful than the owner. A container can:

- Read, but not change, public properties in contained components
- Call methods on composed components
- Listen for some, but not necessarily all, events bubbled up by components that it contains.

Custom App

See App.

Custom Object

Custom records that allow you to store information unique to your organization.

D

Developer Edition

A free, fully-functional Salesforce organization designed for developers to extend, integrate, and develop with the Lightning Platform. Developer Edition accounts are available on developer.salesforce.com.

Development Environment

A Salesforce organization where you can make configuration changes that will not affect users on the production organization. There are two kinds of development environments, sandboxes and Developer Edition organizations.

E

No Glossary items for this entry.

F

No Glossary items for this entry.

G

Getter Methods

Methods that enable developers to display database and other computed values in page markup.

Methods that return values. See also Setter Methods.

Global Variable

A special merge field that you can use to reference data in your organization.

A method access modifier for any method that needs to be referenced outside of the application, either in the SOAP API or by other Apex code.

H

No Glossary items for this entry.

I

ID

See Salesforce Record ID.

Instance

The cluster of software and hardware represented as a single logical server that hosts an organization's data and runs their applications. The Lightning Platform runs on multiple instances, but data for any single organization is always stored on a single instance.

Integrated Development Environment (IDE)

A software application that provides comprehensive facilities for software developers including a source code editor, testing and debugging tools, and integration with source code control systems.

J

No Glossary items for this entry.

L

List View

A list display of items (for example, accounts or contacts) based on specific criteria. Salesforce provides some predefined views.

In the Agent console, the list view is the top frame that displays a list view of records based on specific criteria. The list views you can select to display in the console are the same list views defined on the tabs of other objects. You cannot create a list view within the console.

Lightning Platform

The Salesforce platform for building applications in the cloud. Lightning Platform combines a powerful user interface, operating system, and database to allow you to customize and deploy applications in the cloud for your entire enterprise.

Local Name

The value stored for the field in the user's or account's language. The local name for a field is associated with the standard name for that field.

M

No Glossary items for this entry.

N

Namespace

In a packaging context, a one- to 15-character alphanumeric identifier that distinguishes your package and its contents from packages of other developers on AppExchange, similar to a domain name. Salesforce automatically prepends your namespace prefix, followed by two underscores ("___"), to all unique component names in your Salesforce organization.

O

Object

An object allows you to store information in your Salesforce organization. The object is the overall definition of the type of information you are storing. For example, the case object allow you to store information regarding customer inquiries. For each object, your organization will have multiple records that store the information about specific instances of that type of data. For example, you might have a case record to store the information about Joe Smith's training inquiry and another case record to store the information about Mary Johnson's configuration issue.

Owner

The owner is the component that owns the template. The owner controls all the composed components that it contains. The owner can:

- Set public properties on composed components
- Call methods on composed components
- Listen for any events fired by the composed components

P

No Glossary items for this entry.

Q

No Glossary items for this entry.

R

Record

A single instance of a Salesforce object. For example, "John Jones" might be the name of a contact record.

S

Salesforce Record ID

A unique 15- or 18-character alphanumeric string that identifies a single record in Salesforce.

Setter Methods

Methods that assign values. See also Getter Methods.

Standard Object

A built-in object included with the Lightning Platform. You can also build custom objects to store information that is unique to your app.

T

No Glossary items for this entry.

U

Unit Test

A unit is the smallest testable part of an application, usually a method. A unit test operates on that piece of code to make sure it works correctly. See also Test Method.

User Acceptance Testing (UAT)

A process used to confirm that the functionality meets the planned requirements. UAT is one of the final stages before deployment to production.

V

No Glossary items for this entry.

W

No Glossary items for this entry.

X

No Glossary items for this entry.

Y

No Glossary items for this entry.

Z

No Glossary items for this entry.

INDEX

A

- a [72](#)
- adapters [148–154](#), [156–157](#), [159](#)
- Apex [89](#)
- Aura
 - facets [117](#)

B

- Boolean properties [35](#)
- Bundles [123](#)

C

- Callback method [146](#)
- callbacks [52](#), [55–57](#), [59](#)
- Component bundles [123](#)
- Component models
 - Aura component [117](#)
 - Lightning web component [117](#)
- Component naming schemes [117](#)
- Components
 - Aura events [78](#)
 - bundle [13](#), [17](#)
 - compose [39](#)
 - create [13](#)
 - creating events [71](#)
 - CSS [16](#)
 - data binding [40](#), [64](#)
 - dispatching events [72](#), [78](#)
 - event listener [72](#)
 - event propagation [76](#)
 - events [68–70](#), [76](#)
 - firing events [72](#), [78](#)
 - folder [13](#), [17](#)
 - handling events [72](#)
 - HTML [14](#), [23](#)
 - JavaScript [14](#), [46](#), [68–72](#), [76](#), [78](#)
 - JavaScript properties [25](#), [29–32](#), [34](#), [37](#)
 - namespace [17](#)
 - Properties [25](#), [29–32](#), [34](#), [37](#)
 - semantic events [70](#)
 - sending data [69](#)
 - sending events [72](#), [78](#)
 - sending notifications [69](#)
 - slots [42](#)
 - tests [17](#)

- Composition
 - slots [42](#)
- Conditional [23](#)
- configuration [4](#)
- connectedCallback [56](#)
- constructor [55](#)
- Create an App [18](#)
- create record [111](#), [154–155](#), [158](#)
- CSS [16](#), [26–27](#), [119](#)
- current user [63](#)
- CustomEvent [69](#)

D

- Data
 - Salesforce [82](#), [86](#)
- Data binding
 - Aura [131](#)
 - Lightning Web Components [131](#)
- Debug
 - JavaScript [141](#)
- Debugging [140](#)
- Definition [13](#)
- delete record [155](#)
- Dev Hub [4](#)
- disconnectedCallback [56](#)
- DOM [44](#)
- DOM access [95](#)

E

- errorCallback [59](#)
- ES6 Modules [49](#), [94](#)
- ESLint [160](#)
- Event data [69](#)
- Events
 - Aura [78](#)
 - bubbling [76](#)
 - capture [76](#)
 - container [78](#)
 - creating [71](#)
 - data [69](#)
 - dispatching [72](#), [78](#)
 - extending [70](#)
 - firing [72](#), [78](#)
 - handling [72](#)
 - listening for [72](#)
 - payload [69](#)

Events (*continued*)
 propagation 76
 semantic 70
 sending 72, 78
 simple 69
 structured 70

Extending Event 70

F

Facets 117

G

Getter properties 25, 37
GitHub 6
global value provider 60, 62–63
GVP 60, 62–63

H

HTML 14, 23

I

install 4, 6–7
Interoperability layer
 component naming schemes 117

J

JavaScript
 Aura events 78
 creating events 71
 dispatching events 72, 78
 ES6 modules 120
 event bubbling 76
 event capture 76
 event data 69–70
 event listener 72
 event propagation 76
 events 68–70, 76
 extending events 70
 firing events 72, 78
 getter properties 25
 handling events 72
 HTML attributes 37
 methods 46
 notifications 69
 private properties 37
 properties 29
 property names 30
 public properties 31
 reactive properties 30, 32, 34

JavaScript (*continued*)
 restricted Salesforce globals 96
 secure wrappers 95
 security 94
 sending events 72, 78
 share 120
 share code 49
JavaScript API 95
JavaScript support 4
jest 135, 137, 139

L

label 62
Lifecycle hook 146
Lifecycle hooks 52, 55–57, 59
Lightning App Builder
 creating a width-aware component 113
Lightning Design System 26
Lightning Experience 98, 110
Lightning pages 110
Lightning Testing Service 133–134
Lightning Web Components 1–3, 9
lightning-flexipage-service 113
lint 160
Locker Service
 unsupported browsers 97
LockerService
 DOM access 95
 global references 95
 restricted Salesforce globals 96
 secure wrappers 95
 unsupported browser APIs 96

M

Metadata 15, 89
Methods 46
Migrate
 Apex 131
 attributes 125
 base components 128
 bundles 123
 conditionals 126
 CSS 130
 events 129
 expressions 126
 facets 127
 initializers 127
 iterations 125
 markup 124

Migrate (*continued*)

strategy [123](#)

Modules [49](#), [94](#)

N

Namespaces [17](#)

navigation [99](#)

Navigation

Page Definitions [106](#)

Notifications [69](#)

P

page reference [88](#), [147](#)

PageReference [99](#)

Private properties [37](#)

project [7](#)

Properties

boolean [35](#)

names [30](#)

Public properties [31](#)

R

Reactive properties [30](#), [32](#), [34](#)

record

create [111](#), [154–156](#), [158](#)

delete [155](#)

update [111](#), [155–156](#), [158](#)

RecordInput [155–156](#), [158](#)

renderedCallback [57](#)

repository [6](#)

resource [60](#)

Restricted Salesforce globals [96](#)

S

Salesforce Data [82](#), [86](#)

Salesforce DX [4](#)

Salesforce for Android [98](#), [110](#)

Salesforce for iOS [98](#), [110](#)

Salesforce for mobile web [98](#), [110](#)

Salesforce Lightning Design System [26](#)

sample code [6](#)

Secure wrappers

JavaScript API [95](#)

Security [94](#)

set up [4](#), [6–7](#)

Setup [99](#)

Shadow DOM [44](#), [95](#)

Share code [49](#)

SLDS [26](#)

Slots [42](#)

static resource [60](#)

T

testing [133–135](#), [137](#), [139](#)

tooling [4](#)

U

unit tests [133–135](#), [137](#), [139](#)

Unsupported browser APIs [96](#)

update record [111](#), [155–156](#), [158](#)

URL [88](#), [147](#)

V

validation [160](#)

Visual Studio Code [4](#)

W

web components [1–3](#), [9](#)

Width-aware Lightning component [113](#)

wire service [148–154](#), [156–157](#), [159](#)