Experience Delivery (Beta)

Boost Site Performance and Scalability

Salesforce, Summer ’26, Version 66.0

@salesforcedocs

Last updated: Jun 16, 2026

Important: For sites created with the Build Your Own (LWR) template, Experience Delivery is a pilot or beta service that is subject to the Beta Services Terms at Agreements - Salesforce.com or a written Unified Pilot Agreement if executed by Customer, and applicable terms in the Product Terms Directory. Use of this pilot or beta service is at the Customer's sole discretion.

Important: Experience Delivery (Beta) is being discontinued in October 2026. For more information, see Experience Delivery (Beta).

© Copyright 2000–2026 Salesforce, Inc. All rights reserved. Salesforce is a registered trademark of Salesforce, Inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

CONTENTS

Overview 4
Before You Begin 5
Beta Limitations and Behavior Differences 5
Enable Experience Delivery 7
Disable Experience Delivery 8
Make Your Components Server-Side Ready 9
What Is Server-Side Rendering? 9
Islands Architecture and Component Hydration 9
Verify Component Trees for SSR by Using lwr audit 10
Configure a Component for SSR 11
Identify Non-Portable Code with the ESLint Plugin 12
Use Light DOM or Native Shadow with SSR 12
Protect Your Code from Accessing General Browser APIs 14
Customize Slotting Behavior 15
Work with Scoped Modules Limitations 15
Import Non-Portable Modules Dynamically 16
Remove Host Element Mutations 17
Use Supported Base Components 18
Update Base Lightning Components Styling 18
Fetch Data by Using Data Providers 20
Use Islands Architecture When Implementing SSR 23
Configure a Page for Islands Architecture 23
Determine Island Boundaries and Capabilities 23
Configure a Portable Component for SSR 24
Islands Architecture Considerations 24
Test and Publish Your Site 25
Test Your Components 25
Debug with the SSR Playground 25
Use the SSR Test Runner 26
Manually Verify SSR on a Component 29
Publish Your Site 29
Use Debug Mode for Client-Side Debugging 30
Validate the Network Response of the Published Site 30
Appendix A: Server Renderable Base Components 32
Appendix B: Scoped Modules 33
Appendix C: Provide Your Source Code to Salesforce 34

Overview

Boost the performance and scalability of your LWR sites by using Experience Delivery, our powerful new infrastructure for hosting LWR sites. Along with subsecond page load times, this new infrastructure provides improved security and search engine optimization (SEO) to ensure that your site meets your customers' demands.

Existing LWR sites use client-side rendering (CSR), meaning that all the HTML, JavaScript, CSS, and assets that make up the page are downloaded to the client before being rendered in the browser.

By contrast, Experience Delivery uses server-side rendering (SSR) and a dedicated content delivery network (CDN) to render the page on the server and then cache it in the CDN. This approach provides optimal site performance with page load times up to 60% faster, which leads to increased conversions and lower bounce rates.

Experience Delivery also:

- Enhances SEO by rapidly serving fresh data to web crawlers and bots
- Scales to ensure consumer-grade performance for high-traffic LWR sites
- Provides distributed denial-of-service (DDoS) protection, managed firewall rules, and managed rate-limiting rules for added security

Before You Begin

In Experience Cloud, Experience Delivery is available as a beta feature that you must enable. To use Experience Delivery in Experience Cloud, you need:

- A production org in Enterprise, Performance, or Unlimited Editions with Digital Experiences enabled. Experience Delivery is unsupported in Developer Edition and scratch orgs.
- A new or existing LWR site based on the Build Your Own (LWR) template. LWR and enhanced LWR sites are supported.
- Experience in:
    - Building LWR or enhanced LWR sites with Experience Builder
    - Developing custom Lightning web components that are server-side ready
    - Working with Salesforce DX

Beta Limitations and Behavior Differences

Before you begin, keep the current beta limitations and behavior differences for the Build Your Own (LWR) template in mind.

Unsupported features:

- Custom sitemaps, canonical URLs, and SEO-friendly URL slugs
- Authentication for enhanced domains
- Identity providers for custom domains
- Component variations
- Branding and themes
- Mobile Publisher
- Site archiving
- Login IP range restrictions by user profile
- Down for maintenance
- User-specific modules

Behavior differences:

- Sites hosted on Experience Delivery that use enhanced domains require the Salesforce content delivery network by default. If you choose to turn the CDN off for your org, you must use a custom domain and custom URL to launch your Experience Delivery site.
- Static assets that are available on authenticated pages are cached publicly and therefore potentially available to unauthenticated guest users.
- Unlike sites that aren't hosted on Experience Delivery, when you set up URL redirects, you must publish the site before they take effect.
- Only 20 languages are supported currently.
- Because SEO-friendly URL slugs are unsupported, when you enable Experience Delivery for a site that uses them, the system replaces the friendly URLs of record pages with the record ID. For example, the system replaces a URL such as https://mysite.com/account/portal-acc, which uses the format /account/:urlName, with

https://mysite.com/account/001SG000006px5pYAA/portal-account, which uses the format /account/:recordId/:recordName.
And any bookmarked or hardcoded URLs that use the friendly format are redirected to the record ID format.

- To use a custom domain in your sandbox, you must enable the domain in your production org.

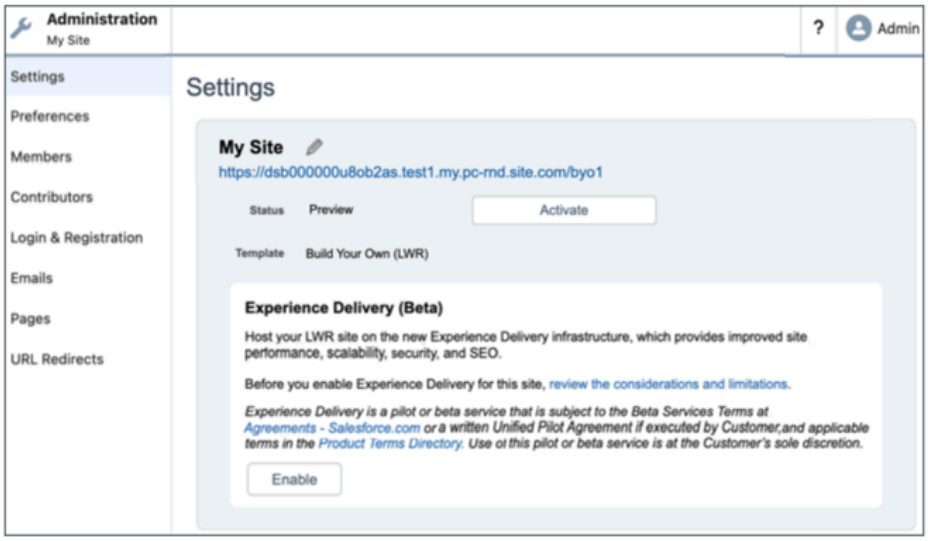
Enable Experience Delivery

In Experience Cloud, you can enable Experience Delivery for new or existing LWR and enhanced LWR sites that are based on the Build Your Own (LWR) template. You enable Experience Delivery at the site level.

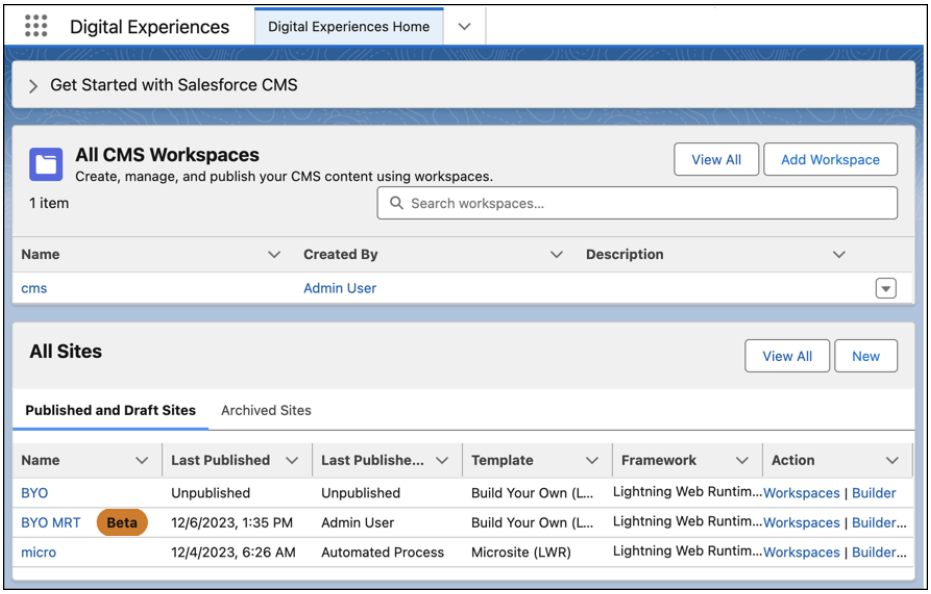
Important: Experience Delivery (Beta) is being discontinued in October 2026. For more information, see Experience Delivery (Beta).

To enable Experience Delivery for an LWR site in Experience Cloud:

1. In the Digital Experiences app, or in Setup under Digital Experiences | All Sites, click **Workspaces** next to the site that you want to configure.
2. On the Settings page of the Administration workspace, click **Enable**.



In the All Sites list, a Beta badge is shown next to sites that are enabled for Experience Delivery.



Disable Experience Delivery

When you disable Experience Delivery, the site is automatically published and hosted on our standard infrastructure. Ensure that any pending site changes are ready to be published before you disable the feature.

1. In the Digital Experiences app, or in Setup under Digital Experiences | All Sites, click **Workspaces** next to the site that you want to configure.
2. On the Settings page of the Administration workspace, click **Disable**.

Make Your Components Server-Side Ready

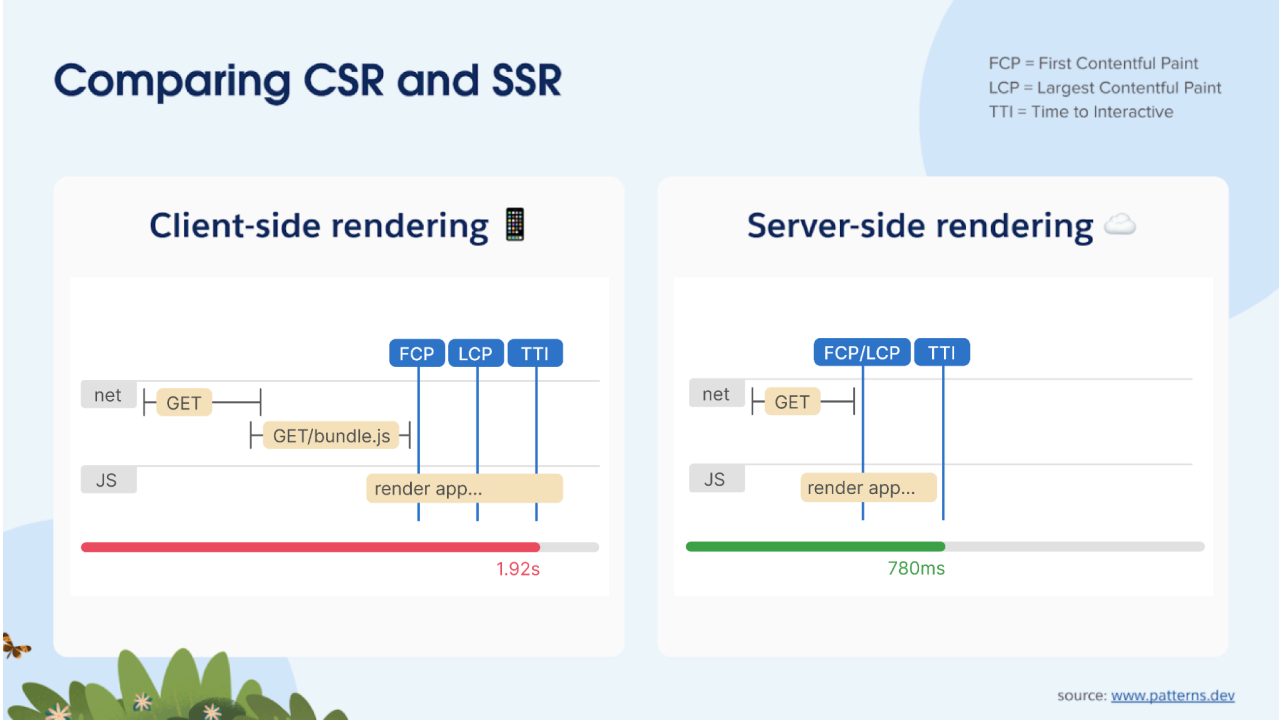
Experience Delivery can greatly improve your site load times. Various factors contribute to the improvement, but one of the most important ones is server-side rendering (SSR).

Tip: For the latest information on configuring components for SSR, check out Server-Side Rendering in the *Lightning Web Runtime on Node.js Guide*.

What Is Server-Side Rendering?

With client-side rendering (CSR), all the HTML, JavaScript, CSS, images, and other assets that make up the page are downloaded to the client, and then the computation required to produce the rendered page is done within the browser.

With SSR, the browser doesn't need to wait for all the JavaScript to download and execute before displaying component markup. Instead, computation is moved to the server, and the resulting page is cached in the CDN for subsequent visitors. This approach results in faster time-to-content, and it makes the content accessible to search engine crawlers, which improves SEO.



Islands Architecture and Component Hydration

With server-side rendering, HTML is rendered on the server and then sent to the client along with any JavaScript needed to hydrate it. Hydration refers to the process of adding interactivity to the component on the client side.

However, there's a performance cost associated with loading and executing excess JavaScript. To counteract this issue, with islands architecture, you can create *islands* of interactivity on a server-rendered page. So rather than an all-or-nothing approach where SSR or CSR is controlled at

the page level, you have granular control over which custom components require hydration, with the remainder of the page being static HTML.

> **Note**: If your component contains a base component and you're unsure if it's server-side renderable, we recommend using the `lightning__ServerRenderableWithHydration` capability tag. See Use Supported Base Components.

For a more detailed description of islands architecture, go to https://www.patterns.dev/vanilla/islands-architecture.

For this beta release, you can create custom components that use:

- **SSR with hydration**–Components render on the server side and get hydrated on the client side. Suitable for interactive page elements, such as a carousel.
- **SSR only**–Components render on the server side. Suitable for static page elements that don't require hydration, such as images.
- **CSR only**–Components render on the client side. Suitable for components that rely on client-side APIs and data that can't be rendered on the server side.

A component must be hydrated if it contains dynamic content or is interactive, including:

- Overriding the render method of `LightningElement`
- Implementing the `renderedCallback` lifecycle hook. (This function isn't called during SSR.)
- Using dynamically bound text or attributes that can change on the client, for example, `<h2>{dynamicText}</h2>`
- Attaching events
- Subscribing to wires that trigger component changes after the initial client-side render
- Composing other components that must be hydrated, for example, `<template><c-form></c-form></template>`

See Also:

- Use Islands Architecture When Implementing SSR

Verify Component Trees for SSR by Using lwr audit

Experience Delivery sites create islands for a page by processing the SSR capabilities of the page's components. For island creation to succeed, your nested components must follow these rules.

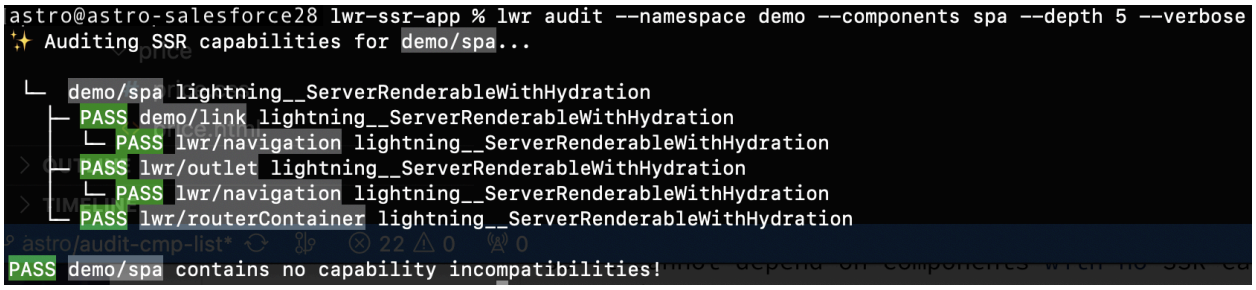
- CSR-only components can contain any type of component.
- Hydrated compounds contain SSR-only components or other hydrated components.
- SSR-only components contain other SSR-only components.

To quickly check if your components follow these rules, use the `lwr audit` command on the LWR command line interface (CLI). You can specify which components to audit on the command line.

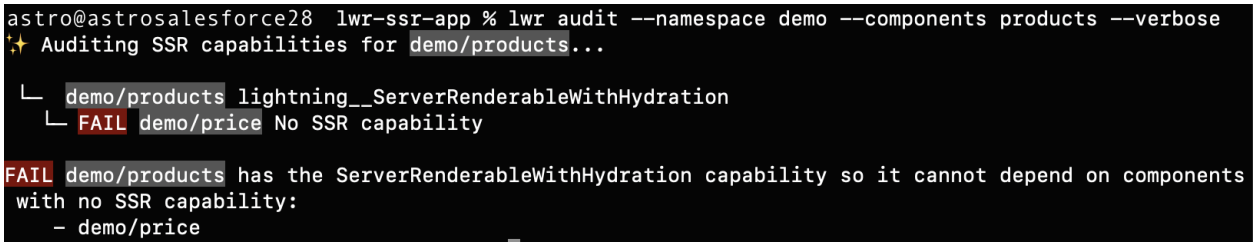
```
None
# audit all components in an SFDX project
ls force-app/main/default/lwc | xargs lwr audit --components
```

```
# audit all components for a given namespace in a LWC or LWR project
ls src/modules/<namespace> | xargs lwr audit --namespace <namespace> --components
# see the list of supported options for the audit command
lwr audit --help
```

For each specified component and all the components in its component tree, `lwr audit` prints the capability assigned to each component and whether or not it's valid for the tree.



If a component fails the SSR audit, review the failure message printed on the command line and assign a valid capability to it.



Configure a Component for SSR

To successfully render a component on the server side:

- The functions that execute during SSR must be **portable**.
- The functions that execute during SSR must be **synchronous**.
- The component must use **light DOM** (recommended) or **native shadow DOM**.

A component is portable if it can run without browser APIs. When SSR runs, the LWC framework executes these functions for a given component and its imported dependencies.

- constructor
- connectedCallback
- getters
- setters
- any other functions called by these functions

Your components can't have any dependencies on browser APIs because the server isn't a browser. If your component uses browser APIs, you must modify the component to ensure it doesn't assume that those APIs are always available.

Here are a few examples of non-portable code and objects.

- `window`
- `document`
- selector functions (`template.querySelector`, `template.querySelectorAll`, ...)
- classList
- JavaScript eventing

Any asynchronous code doesn't complete during SSR because SSR runs in a single synchronous pass. Asynchronous code includes:

- Promises
- async/await
- Dynamic imports

The fastest way to get your components ready for SSR is to use a combination of:

- SSR playground–Identifies SSR-related issues in your Lightning web components
- Local Dev (Beta)–Enables you to run and debug at the site-level in a local production-like environment
- SSR test runner–Guarantees the correctness of your SSR behavior

The next sections describe how to make your components portable and synchronous for SSR.

Identify Non-Portable Code with the ESLint Plugin

The ESLint Plugin for LWC helps to determine whether your components follow best practices on APIs or methods usage. Specifically, use the SSR preset to help make your components server-side renderable.

For example, here are several rules to identify non-portable components.

- Disallow access to global browser APIs during SSR
- Disallow access of properties on `this` during SSR

These rules prevent usage of browser APIs like DOMParser and DocumentFragment in `connectedCallback`, and in methods called from `connectedCallback` or anywhere when SSR is performed.

Additionally, the plugin can help you identify non-portable utilities and libraries. If a library isn't portable, your page doesn't load correctly during SSR.

Use Light DOM or Native Shadow with SSR

When working with SSR, we recommend that your components use light DOM (recommended) or native shadow. Synthetic shadow DOM isn't supported due to the limitations on browser API usage. SSR islands are rendered in native shadow by default.

Note: To convert components from native shadow to light DOM, we recommend that you use the lwc-codemod script. We recommend native shadow only when you need encapsulation and you don't have a versioning strategy for making breaking changes to your components'

DOM structure. For example, use native shadow if you're deploying your components in a managed package.

While shadow DOM encapsulates a component's internals and its styling, light DOM eases third-party integrations, such as with Google Analytics, and enables global styling. For accessibility, light DOM also enables referencing of element IDs across components because components in light DOM aren't encapsulated.

Light DOM ensures that LWC renders regular HTML markup instead of creating a native web component. The markup is also referred to as light DOM because it isn't contained within a shadow root.

To enable light DOM on your Lightning web component, use the `renderMode` static property.

```
JavaScript
import { LightningElement } from 'lwc';
export default class Heading extends LightningElement {
    static renderMode = 'light';
    @api text;
}
```

In your template, use the `lwc:render-mode` directive.

```
None
<template lwc:render-mode='light'>
    <h2 class='global-style'>{text}</h2>
</template>
```

With light DOM, the component content is attached to the host element instead of its shadow tree. You can access the component markup just like any other content in the document host, providing similar behavior to content that's not bound by shadow DOM. Specifically, light DOM supports third-party integrations and global styling. For more information, see the Light DOM documentation.

Slotted content is content that you pass into a slot by using the `<slot>` element. `<slot>` isn't rendered in light DOM. If other parts of your code depend on attributes or event listeners of a `<slot>` element, a compiler error is returned. For example, the `slotchange` event and `::slotted` CSS pseudo-selector aren't supported because `<slot>` doesn't render in the DOM.

Note: Light DOM slots can't use the `slot` attribute to forward slot content into another named slot. Currently, light DOM `<slot>` elements only support forwarding slotted content into the default slot. See the known issue article.

Use native shadow DOM with SSR for encapsulated development, such as when you make changes to a subset of components without impacting other parts of your app or your customer environments. With native shadow, you can improve custom styling by enabling custom themes on your page. You also can use the ::slotted and ::part pseudo functions.

To enable native shadow on a component, use the `shadowSupportMode` static property.

```
JavaScript
import { LightningElement } from 'lwc';
export default class Heading extends LightningElement {
    static shadowSupportMode = 'native';
}
```

In browsers that support native shadow, this component renders in native shadow. Alternatively, use the lwc-codemod script to convert synthetic shadow components to use native shadow.

See Also:

- *Dreamforce 2023 Session:* Experience Delivery - Improve LWCs Load Time with Server Side Rendering
- *Salesforce Developers Blog*: Get Your LWC Components Ready for Native Shadow DOM in Spring '24

Protect Your Code from Accessing General Browser APIs

Rendering components on the server means that as a component author, you don't have access to certain general browser APIs, such as window, document, or querySelectors.

If you're performing these operations during any of the component's lifecycle events, you must protect your code so that it becomes portable.

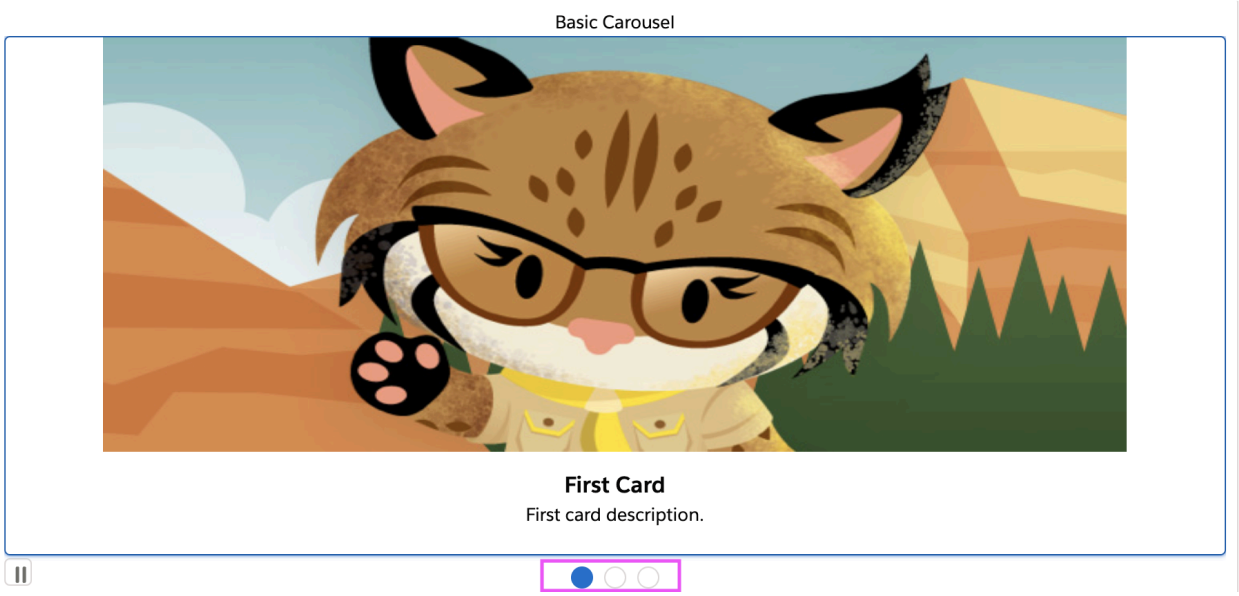
To guard non-portable code, use the `import.meta.env.SSR` boolean. For example:

```
JavaScript
export default class App extends LightningElement {
    connectedCallback() {
        // guard usage of the window object so it does not throw during SSR
        if (!import.meta.env.SSR) {
            window.addEventListener('error', (evt) => {
                console.error(`⚠️ Uncaught error: ${evt.message}`);
            });
        }
    }
}
```

Customize Slotting Behavior

Slotted content renders as expected during SSR. However, native browser built-in events like `slotchange` can't fire during SSR because they're rendered outside of a browser context. In this beta release, your `slotchange` components' behavior can change.

For example, a `slotchange` event determines the number of elements to display in this carousel component. The carousel is outlined in blue, and a pagination component is outlined in pink. The pagination component acts as a control for the number of images being displayed, and in this example it shows a count of 3.



The pagination component logic runs during the `slotchange` event. Because the `slotchange` event doesn't happen during an SSR flow, the first initial render only shows a count of 1.

When the component is rendered in the browser, the `slotchange` event fires and hydration updates the count to 3.

When these browser-specific events occur, the webpage can sometimes display a small "flash" effect. You can reduce the visibility of these effects by adding placeholders or loading messages to your code. This approach is useful when logic takes a long time to run on the client, like a REST or GraphQL data fetch.

Work with Scoped Modules Limitations

Scoped modules provide functionality specific to users, sites, or orgs. Org and site-specific modules are frozen, which means that their values are immutable at publishing time. User-specific modules are evaluated on the client without any server calls. For a list of frozen modules, see Appendix B.

User-specific modules include:

- `@salesforce/user`
- `@salesforce/userPermission`
- `@salesforce/customPermission`

User-specific modules are considered “live scoped modules” because they’re mutable and can change independently of publishing. User-specific modules generally can’t be cached. When a user authenticates, the values are re-fetched on the client.

SSR always runs as a guest regardless of the user requesting the page on a mobile or as an authenticated user. For example, `@salesforce/userisGuest` is always true and `@salesforce/client/formFactor` is always Large on desktop.

If you import `@salesforce/user/isGuest`, it resolves to `true` during SSR regardless of the actual authentication status. However, when the page is hydrated on the client, it resolves to the correct value.

Rendering as a guest on the server and rehydrating as an authenticated user on the client can cause performance and rendering issues. For example, the SSR of content as a guest on the server can display something completely different than the page displayed for the authenticated user. If you use modules that require personalized content, we recommend following these guidelines so that your content renders and hydrates successfully.

- During SSR, render an empty placeholder element. Then, render something more appropriate during CSR.
- Use a CSR-only island to render only on the client.

Import Non-Portable Modules Dynamically

To ensure that non-portable modules don’t get processed on the server, import them dynamically in your component.

For example, you’re importing a portable and a non-portable module in your component.

JavaScript

```
import { portableApi } from 'my/library';
import { nonPortableApi } from 'some/library';
export default class Cmp extends LightningElement {
    connectedCallback() {
        portableApi();
        nonPortableApi();
    }
}
```

To make the imports compatible with SSR, use the async/await function because we don’t want the non-portable code to execute on the server. Also, guard non-portable code by using the `import.meta.env.SSR` boolean.

JavaScript

```
import { portableApi } from 'my/library';
export default class Cmp extends LightningElement {
    async connectedCallback() {
        if (import.meta.env.SSR) {
            portableApi(); // executed during SSR
```

```
        } else {
            const { nonPortableApi } = await import('some/library');
            nonPortableApi(); // NOT reachable during SSR
        }
    }
}
```

Remove Host Element Mutations

Mutating the host element in `connectedCallback()` isn't supported in SSR and CSR. As part of SSR hydration, we validate that the VDOM matches the `HTMLElement` exactly. The VDOM attributes must be equivalent to the `HTMLElement` counterparts for successful validation. If a mutation occurs in a component's `connectedCallback()`, those changes don't appear in the VDOM.

A classic example is attempting to add certain class names to the host element via classList, which is an anti-pattern.

JavaScript
```
// This is an anti-pattern
connectedCallback() {
    // Direct mutation of HTMLElement, such as
    // using this.classList.add or this.setAttribute etc. isn't supported.
    // The 'container' class never appears in VDOM.
    this.classList.add('container');
}
```

Use a semantic element in your template instead. To apply a class:

None
```
<template>
    <form class='container'></form>
</template>
```

To set a class on a parent so that a child renders correctly, we recommend that you wrap the markup within your template tag in a `<div>` tag. Then, implement a corresponding getter in your component's JavaScript.

None
```
<!-- parent.html -->
<template>
  <div class={theClassForChild}>
     <x-child></x-child>
```

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

This example passes in a value to `<c-parent from-outside="parent-class">` by using the `fromOutside` property. The child component renders the `my-child-needs-parent-class` value on the `class` attribute.

JavaScript

```
// parent.js
import { api, LightningElement } from 'lwc';

export default class Cmp extends LightningElement {
    @api fromOutside;
    get theClassForChild() {
        return `my-child-needs-${this.fromOutside}`;
    }
}
```

Note: If you must mutate the host element, such as when there's an external dependency on your internal DOM structure, a workaround is available as a last resort. However, this workaround will be deprecated when a long-term solution is implemented. For more information, see the `validationOptOut` workaround.

Use Supported Base Components

Base components speed up your app development. While most base components support SSR, a small number can't be used for SSR. For example, the `lightning/platformShowToastEvent` module isn't supported for SSR. To communicate feedback after a user action, display a toast by using the lightning/toast module instead.

If you use one of the supported base components, use the `lightning__ServerRenderableWithHydration` capability on your component. For a list of supported base components, see Appendix A.

Update Base Lightning Components Styling

Base Lightning components implement Salesforce Lightning Design System (SLDS) component blueprints and styling. For SSR environments, base Lightning components render in native shadow. While most base components support SSR, a small number can't be used for SSR. See Use Supported Base Components.

Native shadow blocks global styles from being applied to individual components, which means that your existing styling rules sometimes don't work as expected anymore.

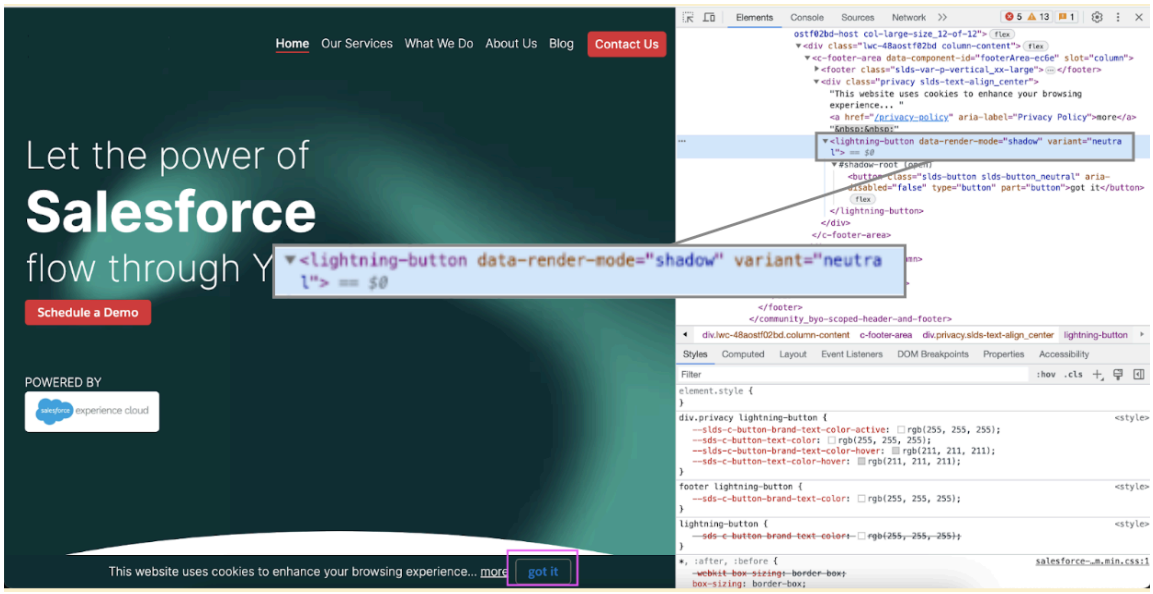
To achieve your styling goals in native shadow, use SLDS styling hooks that are assigned to specific parts of the component with the ::part CSS pseudo function.

Here's an example of how you can modify your existing styling rules. Let's say that you use SLDS styling hooks to customize the text colors on your buttons.

```
None
  div.privacy lightning-button {
    --slds-c-button-brand-text-color-active: rgb(255, 255, 255);
    --sds-c-button-text-color: rgb(255, 255, 255);
    --slds-c-button-brand-text-color-hover: rgb(211, 211, 211);
   --sds-c-button-text-color-hover: rgb(211, 211, 211);
  }
```

In this example, the values aren't applied to lightning-button. The lightning-button component's internals become untargetable because of the native shadow boundary.

The previous example renders this result.

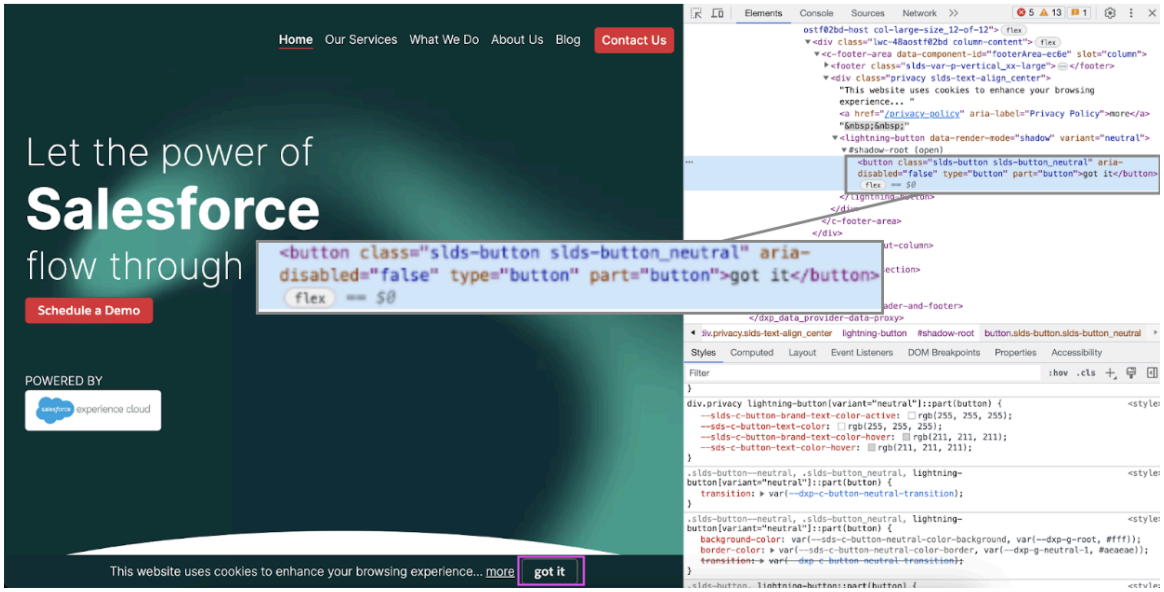


In the footer, the **got it** button text renders in blue instead of white.

To fix this styling issue, apply `::part(button)` with the same rules.

```
None
div.privacy lightning-button::part(button) {
        --slds-c-button-brand-text-color-active: rgb(255, 255, 255);
        --sds-c-button-text-color: rgb(255, 255, 255);
        --slds-c-button-brand-text-color-hover: rgb(211, 211, 211);
        --sds-c-button-text-color-hover: rgb(211, 211, 211);
    }
```

Using `::part(button)` results in the correct styles getting applied to the button.



See Also:

- *Lightning Design System:* *Styling Hooks on SLDS Components*
- *Lightning Design System:* Styling Hooks

Fetch Data by Using Data Providers

SSR runs in a single synchronous pass, so fetching data, which is an asynchronous action, isn't supported in server-side rendered component code. As a result, you can't use tools like `@wire`, `fetch`, and `XMLHttpRequest`.

To fetch data on the server in an Experience Delivery app, use data providers instead. With data providers, you pull data into a view, and you can reference the data in a view by using a data expression. Data providers dynamically populate data expressions with property values.

You can create a data provider from scratch, or you can convert a wire adapter or a public Apex controller into a data provider.

Tip: The Apex Data Providers pilot introduces data providers that act as a gateway to interact with Apex controllers and help preload data onto your site. For more information, see Apex Data Providers (Pilot).

Convert an Apex Wire Adapter to an Apex Data Provider

To access `@wire` data during SSR, convert the Apex wire adapter into an Apex data provider. So if your custom component retrieves data using a direct `@wire` call, modify it to use an `@api` variable instead.

For example, this `c-account` component uses `@wire` to get a value for `name` from the `data` dataset.

```
None
// c/account using @wire
```

```
import { LightningElement, wire } from 'lwc';
import getAccountName from '@salesforce/apex/ApexDataProviderDemo.getAccount';

export default class Account extends LightningElement {
    name;

    @wire(getAccountName)
    processName({ error, data }) {
        if (data) {
            this.name = data;
        }
    }
}
```

To make `c-account` fetch data synchronously, revise it to call `@api` instead of `@wire`.

```
None
// c/account using @api

import { LightningElement, api } from 'lwc';

export default class Account extends LightningElement {
    @api accountName; // matches the attribute key from the metadata below
}
```

Then, incorporate the data provider into the view's metadata.

```
None
{
    "attributes": {
        "accountName": "{!ApexDataProvider.Name}"
    },
    "definition": "c:account",
    "id": "abc-123",
    "type": "component",
    // Add the dataProviders block if it doesn't already exist
    "dataProviders": [
        {
            "definition": "sfdc_cms__apexDataProvider",
            "sfdcExpressionKey": "ApexDataProvider",
            "attributes": {
                "apexClass": "ApexDataProviderDemo",
                "apexMethod": "getAccount",
                "apexParams": {
                    "name": "Example Account"
```

```
                }
            }
        }
    ]
}
```

Enable an Apex Controller to Use the Apex Data Provider

With Apex data providers, you can access data from existing public Apex controllers. An Apex controller is an Apex class that stores information and logic for a group of UI components.

To set up an Apex data provider, annotate at least one method of the Apex class with `@AuraEnabled(cacheable=true scope='global'`. The class can contain only supported input parameters and return type.

For example, this data provider fetches account name data from the `ApexDataProviderDemo#getAccount` Apex controller. It then passes that data to a text block component.

JavaScript

```
public class ApexDataProviderDemo {
    @AuraEnabled(cacheable=true scope='global')
    public static Account getAccount(String name){
        Account account = [SELECT FIELDS(STANDARD) FROM Account WHERE Name
=:name];
        return account;
    }
}
```

You must add this data provider to the `textBlock` component in the payload.

None

```
"definition": "dxp_base:textBlock",
"id": "1ba73c17-b25c-4bda-82bd-c49c5e8c1e9c",
"type": "component",
// If dataProviders block is not present need to add it in the payload
"dataProviders": [
{
"definition": "sfdc_cms__apexDataProvider",
"sfdcExpressionKey": "ApexDataProvider",
"attributes": {
"apexClass": "ApexDataProviderDemo",
"apexMethod": "getAccount",
"apexParams": {
"name": "Example Account"
}}}]
```

Then, to bind textBlock to the data from `ApexDataProviderDemo#getAccount`, use the expression `{!ApexDataProvider.Name}` on the text field of the text block component.

Use Islands Architecture When Implementing SSR

Use islands architecture to create islands of component interactivity on a server-rendered page.

Configure a Page for Islands Architecture

To opt into SSR with islands architecture, the theme layout component of a page must include the `lightning__ServerRenderable` capabilities tag. In the default pages of the Build Your Own (LWR) template, this capability is enabled on all standard pages.

Tip: If you need an entire route to be client-side rendered, we recommend creating a custom theme layout component that doesn't include the `lightning__ServerRenderable` capabilities tag. See Create Custom Layout Components in the *LWR Sites for Experience Cloud Guide*.

Determine Island Boundaries and Capabilities

An island boundary is determined by the first component encountered in the component tree that's set to SSR with hydration or to CSR only. If a component is set to SSR only, it simply becomes part of the page HTML returned by the server, as nothing is required on the client side.

When nesting components, islands capability is determined by the least SSR-capable component within the island boundary. For example:

- If a parent component is set to SSR with hydration and its child component is set to CSR only, both components are rendered on the client side.
- If a parent component is set to CSR only, the parent and its child components are rendered on the client side.
- If a parent component is set to SSR with hydration and its child component is set to SSR only, both components are hydrated.

For more information, see Nested Components and Precedence in the *Lightning Web Runtime on Node.js Guide*.

Configure a Portable Component for SSR

To enable a portable component for SSR, add one of these capability tags to the component's configuration file.

- `lightning__ServerRenderable` enables a component for SSR without hydration. Use this capability only if you don't expect any updates or interactivity from your components after they're rendered.
- `lightning__ServerRenderableWithHydration` enables a component for SSR with hydration. Use this capability if your component contains any dynamic content or if it's interactive.
- CSR is the default capability, so it doesn't require a tag.

Important: If a template contains any child components that aren't portable, then the parent component also isn't portable. Don't include these capabilities on a non-portable component because the entire page fails to render with a 500 error from the server. See Verify Component Trees for SSR by Using lwr audit.

```
None
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
   <apiVersion>58.0</apiVersion>
   <capabilities>
       <capability>lightning__ServerRenderableWithHydration</capability>
   </capabilities>
</LightningComponentBundle>
```

Tip: To use a component in Experience Builder, remember to include the `targets` tag with the appropriate values, such as `lightningCommunity__Page` or `lightningCommunity  Default`.

Islands Architecture Considerations

- Components that use dynamic component visibility are considered CSR-only regardless of the capability tag in the component metadata.
- Expressions used for dynamic data aren't considered when calculating island boundaries or capabilities.
- During SSR, all data requests are made as if the user hasn't logged in. As a result, auth-related scoped modules and data are resolved in a guest user context. Authenticated data is fetched during hydration, so a component only updates with authenticated data on the client side.

Test and Publish Your Site

Before you test components, make sure that you reviewed the Make Your Components Server-Side Ready section.

Test Your Components

Test your components for SSR in these ways.

- SSR playground
- SSR test runner
- Manual Testing

Debug with the SSR Playground

The SSR playground enables you to render, debug, and experiment with individual components and their children in CSR and SSR modes.

Run the playground commands from a directory that contains one of these files.

- `lwc.config.json` file
- `package.json` file containing an `lwc` field
- `sfdx-project.json` file that points to a project directory containing LWCs

To open the playground in Chrome, run this command.

```
None
npx -p @lwc/wds-playground playground namespace/component
```

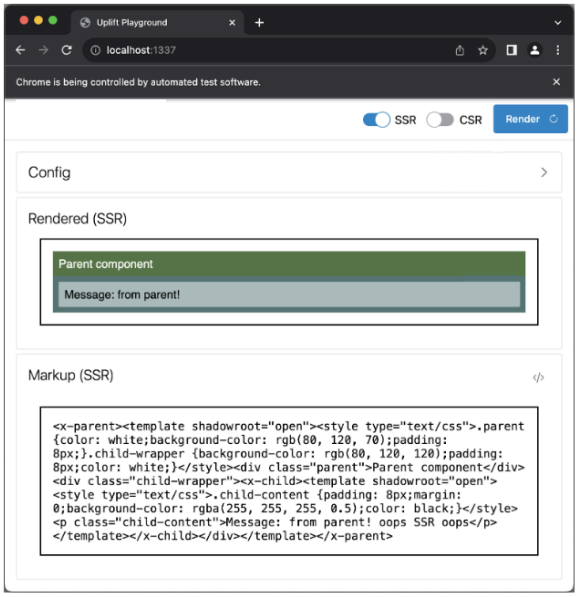
To open the playground in Chrome with DevTools.

```
None
npx -p @lwc/wds-playground playground namespace/component --open --devtools
```

Start at your component tree's "leaves," or components that don't contain other components.

1. Open the component in the SSR playground.
    a. Modify the component props to test important use cases.
    b. Address errors that are reported during the three phases of hydration: SSR, DOM insertion, and rehydration.
    c. Look for and address any visual bugs in your component.
2. Enable the CSR toggle to render your component with SSR and CSR side by side.
3. Use the playground's comparison tool to ensure that the SSR and CSR component instances are visually identical.

4. Enable the layout shift tool in the **Misc** section in config to observe any layout shifts during SSR hydration.



By default, the SSR playground takes your component through three stages of its SSR lifecycle.

- Render the component to HTML markup on the server.
- Render the HTML on the client.
- Hydrate the DOM subtree and associate it with an instance of your component class.

We recommend that you write tests by using the SSR test runner to ensure that your components don't regress as you make changes.

Use the SSR Test Runner

Just like you can use Jest to write unit tests for your LWCs, you can use the SSR test runner to make assertions about your LWCs in SSR-related scenarios.

By default, the SSR tests run in headless Chrome. Unlike Jest tests, the SSR tests run in a full-featured web browser

Use these functions from `@lwc/test-runner`.

- `renderToMarkup` is an asynchronous function that takes the path to your component and the properties that you use for rendering. It returns `Promise<String>`, where the `String` is HTML markup.
- `insertMarkupIntoDom` is an asynchronous function that takes SSR markup, such as `Promise<String>` returned by `renderToMarkup`, as its single argument. It returns `Promise<HtmlElement>`, which is a handle to the root element of your SSR-rendered DOM subtree.
- `hydrateElement` is an asynchronous function that takes a root element, such as `Promise<HtmlElement>` returned by `insertMarkupIntoDom`, and component properties. The properties should be the same as those passed into `renderToMarkup`. This function

returns a `Promise<Boolean>`, where the `Boolean` indicates whether hydration completed without validation errors. If hydration fails, review the errors in the console.

- `expect` is a function that you can use to chain assertions. To make it easier to test your components, it provides all of the assertions from `Chai.expect` in addition to these assertions.
    - `SSRCorrectly`–Checks that all three stages of the SSR lifecycle are performed correctly.
    - `visuallyIdenticalInCSRandSSR`–Performs a pixel match of SSR and CSR components.
    - `noLayoutShifts`–Ensures that the component didn't have any layout shifts when it was hydrated in the DOM.
    - `notMakeDomMutationsDuringSSR`–Ensures that the component didn't make any DOM mutations.

To invoke the test runner, run this command.

```
None
npx -p @lwc/test-runner test-lwcs SPEC_FILE_PATTERN
```

If you use ZSH, surround `SPEC_FILE_PATTERN` in single quotes.

To distinguish the SSR tests from Jest tests, use unique file extensions. For example, if your Jest tests follow the format `COMPONENT_NAME.spec.js`, follow the format `COMPONENT_NAME.spec-ssr.js` for your SSR test files. If you follow this filename format, run your tests like this one.

```
None
npx -p @lwc/test-runner test-lwcs './src/**/*.spec-ssr.js'
```

Tests run in parallel in separate headless Chrome tabs. The tests run fast, and as much as 10,000 tests can complete in under 6 seconds, depending on your hardware.

Here's an example of an SSR test.

```
JavaScript
import {
    // Importing `expect` from the toolkit is subject to change
    expect,
    querySelectorDeep,
    // These three functions contribute heavily to your tests
    renderToMarkup,
    insertMarkupIntoDom,
    hydrateElement,
} from '@lwc/test-runner';

// Instead of importing the component directly, use the path of
// the component's JS file. This value is passed to `renderToMarkup`
// and `hydrateElement` where the component is imported automatically.
const componentPath = import.meta.resolve('./parent.js');

describe('<x-parent>', () => {
    it('is SSR-able and very snazzy', async () => {
```

```
    // Errors thrown during SSR cause the test to fail.
    const markup = await renderToMarkup(componentPath, {});

    // Once you have the raw HTML markup, you can make related assertions.
    expect(markup).to.contain('</x-parent>');

    // Insert that markup into the DOM. It doesn't get attached
    // to an instance of your component class yet.
    const el = await insertMarkupIntoDom(markup);

    // Make assertions about pre-hydrated DOM.
    expect(el).to.haveShadowChild('p.child-content');

    // Finally, hydrate the HTML that was generated on the server and
    // inserted into the DOM in previous steps.
    const hydratedWithSsrDOM = await hydrateElement(el, componentPath);

    // Check that hydration occurred without validation errors.
    expect(hydratedWithSsrDOM).to.be.true;

    // Now that the SSR-generated markup has been inserted and hydrated,
    // the component should behave as though it were originally rendered
    // in the browser with CSR. Check the component to ensure expected behavior.
    expect(querySelectorDeep('p.child-content', el)).to.have.text('Hmm! hello
    from parent');
    });

    // Unless you want to make assertions on output of each individual stage of
SSR,
    // you can use this one-liner to check that DOM was hydrated without any
errors.
    it('SSR correctly', async () => {
        await expect(componentPath, {}).to.SSRCorrectly();
    });

    // Make sure no error is thrown when connected callback lifecycle in executed.
    it('doesnt throw in connected callback', async () => {
        await expect(async () => {
            await renderToMarkup(componentPath, {});
        }).to.not.throwErrorInConnectedCallback();
    });

    // Pixel match rendered SSR and CSR components to check for visual
correctness.
    it('checks that SSR and CSR are visually the same ', async () => {
        await expect(componentPath, {}).to.be.visuallyIdenticalInCSRandSSR();
    });

    // No layout shift was observed during hydration stage.
    it('has no layout shifts', async () => {
        await expect(componentPath, {}).to.have.noLayoutShifts();
    });

    // The component doesn't make any DOM mutation.
    it('makes no DOM mutations', async () => {
        await expect(componentPath, {}).to.notMakeDomMutationsDuringSSR();
    });
```

Manually Verify SSR on a Component

When a component is rendered on a page, you can verify that SSR was successful by looking at the DOM elements.

To determine if an island or component tree implemented SSR successfully:

1. In Chrome, right-click the page and go to **View Page Source**. If you use Firefox or another browser, **View Page Source** can correspond to another option like **Inspect** or **View Selection Source**.
2. Search for some HTML or text from the component. If you find it, then SSR was successful.

In Experience Delivery apps, search for `webruntime-island-container`, which wraps each CSR'ed and hydrated island.

- **CSR-only**–The wrapper has no inner HTML.
- **SSR with hydration**–The wrapper has inner HTML and adata-lwr-props-id attribute.
- **SSR-only (not hydrated)–**Doesn't have a wrapper. Its HTML exists in the page source.

Here are a few examples.

```
None
<!-- CSR-only -->
<webruntime-island-container-x1787601173></webruntime-island-container-x1787601173
>

<!-- SSR with hydration -->
<webruntime-island-container-x1783627592
data-lwr-props-id="lwcprops8de0"><c-cmp><style
type="text/css">...</style><textarea></textarea></c-cmp></webruntime-island-contai
ner-x1783627592>

<!-- SSR-only -->
<c-title><style type="text/css">...</style><h1>Page Title</h1></c-title>
```

See Also:

- *LWR Sites for Experience Cloud Guide*: Create Components for LWR Sites

Publish Your Site

Before publishing your site to make it live, preview the site in Experience Builder to verify that it looks as expected. You can publish your LWR site at any time, even when the site is unchanged.

For LWR sites, if you change your organization's schema or update a component used in an LWR site, you must publish your site to make the changes live. Otherwise, your site can break at runtime.

To publish your changes, click **Publish** on the toolbar in Experience Builder.

Alternatively, you can publish your site by using the CLI command.

```
None
sf community publish --name 'My Site Name'
```

An email notification informs you when your changes go live.

See Also:

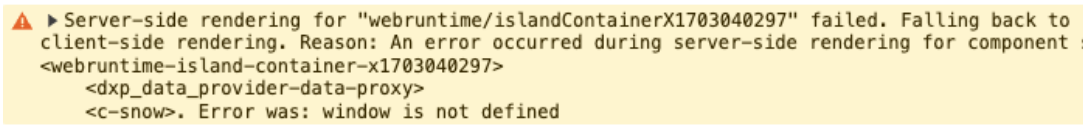
- *Salesforce Help:* Preview Your Experience Builder Site
- *Salesforce Help*: Publish Your Experience Builder Site
- *LWR Sites for Experience Cloud Guide*: New Publishing Model for LWR Sites
- *Salesforce CLI Command Reference*: `community` Commands

Use Debug Mode for Client-Side Debugging

Use debug mode to troubleshoot client-side code in your published site. When you use debug mode, framework and component JavaScript code isn't minified and is easier to read and debug. Debug mode also adds more detailed output for some warnings and errors.

To use debug mode:

1. Open the published site page in your browser and append *?debug* to the URL. For example, *https://www.mysite.com/?debug*.
   Note: For LWR sites hosted on Experience Delivery, the Debug Mode Users setting in Setup isn't supported.
2. Open the browser's developer tools.
3. With the page in debug mode, you can:
   - View unminified JavaScript files to make debugging easier.
   - Locate server-side-rendering error messages by searching for "An error occurred during server-side rendering" and expanding the message.



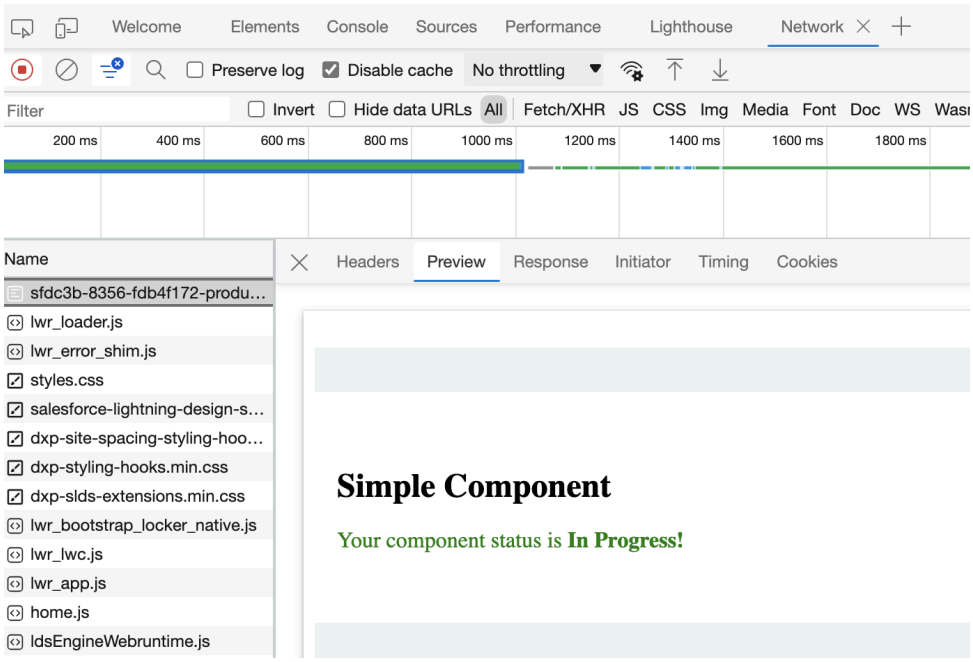
Validate the Network Response of the Published Site

Now that you converted your components to be server-side renderable and published your site to Experience Delivery, you can verify the final output of a given page rather than validating each individual component.

The easiest method is to use the Network Response for your root document in the browser's Developer Tools.

1. Go to your URL.
2. To open the browser's Developer Tools, right-click the page and click **Inspect**, or press the F12 key.

3. Open the Network tab.
4. Ensure that the network is being recorded, as indicated by the red square with a circle icon in the top left of the window.
5. Ensure that the All filter is selected, and reload the page via the browser's refresh button. The network panel shows all the requests coming in.
6. Click the first item in the list, which is the root document with the name of the current URL and type of document.
7. In the side window that opens, click the Preview tab.
   Any content that was server-side rendered shows in that window. If you click the Response tab, you can see the static HTML, JavaScript, and CSS that was delivered as a result of SSR.



Alternatively, you can turn off JavaScript in your browser to review your published site. After turning off JavaScript, refreshing your browser displays only content that's server-side rendered. Disabling JavaScript isn't supported in Preview mode.

Appendix A: Server Renderable Base Components

These base components are server-side renderable or can be rendered on a page with an SSR-enabled component. To use these base components, include the `lightning__ServerRenderableWithHydration` tag in your component's `.js-meta.xml` file.

- lightning-accordion
- lightning-accordion-section
- lightning-alert
- lightning-avatar
- lightning-badge
- lightning-breadcrumb
- lightning-breadcrumbs
- lightning-button
- lightning-button-group
- lightning-button-icon
- lightning-button-icon-stateful
- lightning-button-menu
- lightning-button-stateful
- lightning-card
- lightning-checkbox-group
- lightning-combobox
- lightning-confirm
- lightning-dual-listbox
- lightning-dynamic-icon
- lightning-file-upload
- 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-location
- lightning-input-rich-text
- lightning-layout
- lightning-layout-item
- lightning-menu-divider
- lightning-menu-item
- lightning-menu-subheader
- lightning-modal
- lightning-modal-body
- lightning-modal-footer
- lightning-modal-header
- lightning-pill
- lightning-pill-container
- lightning-progress-bar
- lightning-progress-indicator
- lightning-progress-ring
- lightning-progress-step
- lightning-prompt
- lightning-radio-group
- lightning-relative-date-time
- lightning-rich-text-toolbar-button
- lightning-rich-text-toolbar-button-group
- lightning-select
- lightning-slider
- lightning-spinner
- lightning-tab
- lightning-tabset
- lightning-textarea
- lightning-tile
- lightning-toast
- lightning-toast-container
- lightning-tree
- lightning-tree-item
- lightning-vertical-navigation
- lightning-vertical-navigation-item
- lightning-vertical-navigation-item-badge
- lightning-vertical-navigation-item-icon
- lightning-vertical-navigation-overflow
- lightning-vertical-navigation-section

Base components that aren't listed haven't been evaluated or enabled for SSR yet. See Use Supported Base Components. For base component documentation, see the Component Library.

Appendix B: Scoped Modules

Org and site-specific modules are frozen, which means that their values are immutable at publishing time. User-specific modules are evaluated on the client without any server calls. See Work with Scoped Modules Limitations.

These Salesforce scoped modules are frozen, which means immutable until publish.

- `@salesforce/apex/*`
- `@salesforce/client/formFactor`
- `@salesforce/community/Id`
- `@salesforce/contentAssetUrl/*`
- `@salesforce/i18n/*`
- `@salesforce/label/*`
- `@salesforce/messageChannel/*`
- `@salesforce/resourceUrl/*`
- `@salesforce/schema/*`

This scoped module is frozen and can require another publish if its value is changed.

- `@salesforce/community/basePath`

User-specific modules are mutable and can change independently of publishing. These modules can't be cached. When a user authenticates, the values are re-fetched on the client. User-specific modules include:

- `@salesforce/user`
- `@salesforce/userPermission`
- `@salesforce/customPermission`

These scoped modules are frozen at publish.

- `@salesforce/i18n/lang`
- `@salesforce/site/Id`

For more information on scoped modules, see `@salesforce` modules.

Appendix C: Provide Your Source Code to Salesforce

The Experience Delivery team sometimes requests access to source code and CMS content for testing purposes. Sharing your source code is entirely optional, but it does help the team immensely.

You can use Salesforce CLI commands to export these Metadata API types for the site.

- Network
- CustomSite
- ExperienceBundle (for LWR sites)
  OR
  DigitalExperienceBundle and DigitalExperienceConfig (for enhanced LWR sites)
- StaticResource
- ApexClass
- LightningComponentBundle

Here's an example of a package.xml manifest file.

```
None
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
   <types>
       <members>*</members>
       <name>Network</name>
   </types>
   <types>
       <members>*</members>
       <name>CustomSite</name>
   </types>
   <types>
       <members>*</members>
       <name>ExperienceBundle</name>
   </types>
   <types>
       <members>*</members>
       <name>DigitalExperienceBundle</name>
   </types>
   <types>
       <members>*</members>
       <name>DigitalExperienceConfig</name>
   </types>
   <types>
       <members>*</members>
       <name>StaticResource</name>
```

```
    </types>
    <types>
        <members>*</members>
        <name>LightningComponentBundle</name>
    </types>
    <types>
        <members>*</members>
        <name>ApexClass</name>
    </types>
    <types>
        <members>*</members>
        <name>CustomObject</name>
    </types>
    <version>57.0</version>
</Package>
```

And here's an example of the `sf project retrieve` command.

```
None
sf project retrieve start --manifest manifest/package.xml
```

See Also:

- Salesforce CLI Setup Guide
- Salesforce CLI Command Reference
- *Experience Cloud Developer Guide*: ExperienceBundle for Experience Builder Sites
- *Experience Cloud Developer Guide*: Deploy Your Experience Cloud Site with the Metadata API

Provide CMS Content

To export Salesforce CMS content that's used in the site, see Import and Export Content with Salesforce CMS in *Salesforce Help*.