
Lightning Communities Developer Guide

Version 41.0, Winter '18



CONTENTS

Chapter 1: Get Up to Speed with Lightning Communities	1
Before You Begin	2
What Is Salesforce Lightning?	3
What Is the Lightning Component Framework?	3
Which Lightning Template Do I Use?	5
Chapter 2: Develop Lightning Communities: The Basics	6
Using the Developer Console	7
Configure Drag-and-Drop Components for Community Builder	7
Exposing Component Attributes in Community Builder	10
Tips and Considerations for Configuring Components for Community Builder	11
Supported Lightning Components, Interfaces, and Events for Communities	12
Chapter 3: Customize the Look and Feel of a Lightning Template	14
Update a Template with the Branding Panel	15
Override Template Elements with Custom CSS	15
Migrate CSS Overrides	18
Use Custom Fonts in Your Community	21
Customize the Theme Layout of Your Template	23
How Do Custom Theme Layouts Work?	23
Configure a Custom Theme Layout Component	25
Create Custom Content Layout Components for Communities	28
Configure Swappable Search and Profile Menu Components	29
Standard Design Tokens for Communities	31
Chapter 4: Example: Build a Condensed Theme Layout Component	34
Step 1: Create the Basic Theme Layout Structure	35
Step 2: Define a Tokens Bundle	37
Step 3: Add a Logo Component	37
Step 4: Build a Vertical Navigation Menu	38
Step 5: Build a Custom Search Component	39
Step 6: Add Configuration Properties to the Theme Layout	43
Chapter 5: Develop Secure Code: LockerService and Stricter CSP	45
LockerService in Communities	46
Critical Update for Stricter CSP Restrictions in Communities	46
Chapter 6: Analyze and Improve Community Performance	49
Chapter 7: Connect Your Community to Your Content Management System	54

Contents

Before Using CMS Connect	55
Create a CMS Connection	56
Build a CMS Connect Root Path and Component Paths	58
Set Up Language Mapping in Your CMS Connection	58
Set Up JSON in Your CMS Connection (Beta)	59
Edit a CMS Connection	62
Manage CMS Connections	62
Add CMS Content to Your Community Pages	63
Add CMS Header and Footer Components to Your Community	63
Add CMS Connect (HTML) Components to Your Community Pages	64
Add CMS Connect (JSON) Components to Your Community Pages (Beta)	64
Personalize Your CMS Content	67
CMS Connector Page Code	70
CMS Connect Recommendations for Optimal Usage	75
CMS Connect Examples	76
Example: Connect JSON Content to Your Community	76
Chapter 8: Community Migration, Packaging, and Distribution	79
Migrate Your Community with Change Sets	80
Considerations for Migrating Communities with Change Sets	81
Lightning Bolt Solutions: Build Once, Then Distribute and Reuse	82
Export and Packaging Considerations for Lightning Bolt Solutions	84
Requirements for Distributing Lightning Bolt Solutions	86
Export a Customized Lightning Bolt Solution	87
Export a Customized Lightning Bolt Page	89
Package and Distribute Lightning Bolt Solutions or Pages	90
INDEX	91

CHAPTER 1 Get Up to Speed with Lightning Communities

In this chapter ...

- [Before You Begin](#)
- [What Is Salesforce Lightning?](#)
- [What Is the Lightning Component Framework?](#)
- [Which Lightning Template Do I Use?](#)

Lightning community templates let you create branded spaces where your employees, customers, and partners can connect. Built on the Lightning Component framework, Lightning templates include many ready-to-use features and Lightning components. But the real power of the Lightning Component framework is that you can develop custom Lightning components and features to meet your unique business needs and completely transform the look and feel of your community.

Whether you're a developer, partner, or ISV, this guide helps you create custom Lightning communities, components, theme layouts, and Bolt solutions. You'll also learn how to package solutions and components and distribute them on AppExchange.

Before You Begin

Before you begin developing custom Lightning communities, ensure that you're familiar with developing in Lightning.

You can create Lightning communities and Lightning components using the UI in **Enterprise, Performance, Unlimited,** and **Developer Editions** or a sandbox.

To use this guide successfully, it helps to have:

- An org with Communities enabled
- A new or existing community that's based on the Customer Service (Napili) template or a Lightning Bolt solution
- Experience using Community Builder and the Customer Service (Napili) template
- Experience developing Lightning components and using CSS

Resources for Lightning Development

Unfamiliar with Lightning development? Then check out these resources.

Lightning Component Developer Guide

Think of the *Lightning Component Developer Guide* as your best friend. It's the go-to guide for all things Lightning, and the foundational concepts and approaches it documents form the bedrock of this guide. Think of the *Lightning Communities Developer Guide* as Part 2 in the Lightning development series; it's no use to you until you've familiarized yourself with Part 1.

Lightning Components Basics (Module)

Use Lightning components to build modern web apps with reusable UI components. Learn core Lightning components concepts and build a simple expense tracker app that can be run in a standalone app, the Salesforce app, or Lightning Experience.

Lightning Design System (Module)

Build pixel-perfect enterprise apps using our design guidelines and CSS framework.

Quick Start: Lightning Components (Project)

Create your first component that renders a list of contacts from your org.

Build an Account Geolocation App (Project)

Build an app that maps your accounts using Lightning components.

Build a Lightning App with the Lightning Design System (Project)

Design a Lightning component that displays an account list.

Lightning Components Performance Best Practices (Blog Post)

Learn about Lightning characteristics that impact component performance, and get best practices to optimize your components.

Resources for Communities

Unfamiliar with Communities? Then check out these resources.

Set Up and Manage Salesforce Communities (Guide)

Customize and create communities to meet your business needs.

Using Templates to Build Communities (Guide)

Create branded communities using Lightning templates to interact directly with your customers and partners online.

Expand Your Reach with Communities (Trail)

Learn the tools you need to get started with Salesforce Community Cloud.

CMS Connect Developer Guide (Beta)

Use CMS Connect to embed content from a third-party Content Management System, such as Adobe Experience Manager (AEM), in your Salesforce community. Connect CMS components, HTML, CSS, and JavaScript to customize your community and keep its branding consistent with your website.

Set Up SEO for Your Community (Help)

Have search engines, such as Google™ or Bing®, index your community so that customers, partners, and guest users can easily discover community pages via online searches.

Salesforce Communities Resources (Help)

Stay up to date on other Communities resources.

What Is Salesforce Lightning?


Salesforce Lightning makes it easier to build responsive applications for any device, and encompasses the Lightning Component framework and helpful tools for developers.

Lightning includes these technologies.

- Lightning components accelerate development and app performance. The client-server framework is ideal for use with Communities, in addition to the Salesforce mobile app and Salesforce Lightning Experience.
- Lightning App Builder empowers you to build Lightning pages visually, without code, using off-the-shelf and custom-built Lightning components. You can make your Lightning components available in the Lightning App Builder so administrators can build custom user interfaces without code.
- Community Builder is used to design and build communities using Lightning templates and components. Just like the Lightning App Builder, you can use standard or custom components so that administrators can create community pages with point-and-click customizations.

Some Salesforce products built with underlying Salesforce Lightning technology include:

- The Customer Service (Napili) and Partner Central community templates
- [Lightning Bolt solutions](#) because they're based on the Customer Service (Napili) template
- Lightning Experience
- Salesforce app

 **Note:** You don't need to enable Lightning Experience to use Lightning community templates or develop Lightning components. Lightning communities use the same underlying technology as Lightning Experience, but they're independent of each other.

SEE ALSO:

[Salesforce Help: How Communities Use Lightning](#)

[Lightning Component Developer Guide: Browser Support Considerations for Lightning Components](#)

What Is the Lightning Component Framework?

The Lightning Component framework is a UI framework for developing dynamic web apps for mobile and desktop devices. You can build single-page applications engineered for growth.

The framework supports partitioned, multi-tier component development that bridges the client and server. It uses JavaScript on the client side and Apex on the server side.

The benefits include an out-of-the-box set of components and interfaces, event-driven architecture, and a framework optimized for performance.

Components

Components are the self-contained and reusable units of an app, which represent a reusable section of the UI. They can range in granularity from a single line of text to an entire app.

The framework includes a set of prebuilt components. For example, components that come with the Lightning Design System styling are available in the `lightning` namespace and are known as the base Lightning components. You can assemble and configure the components to form new components in an app. Components are rendered to produce HTML DOM elements within the browser.

A component can contain other components, along with HTML, CSS, JavaScript, Apex controllers, or any other web-enabled code, which enables you to build apps with sophisticated UIs.

The details of a component's implementation are encapsulated. Encapsulation allows the consumer of a component to focus on building an app, while the component author can continue to innovate and make changes without breaking consumers' apps. You configure components by setting the named attributes that they expose in their definition. Components interact with their environment by listening to or publishing events.

Events

Many languages and frameworks use event-driven programming, such as JavaScript and Java Swing. Handlers that you write respond to interface events as they occur.

A component registers that it might fire an event in its markup. Events are fired from JavaScript controller actions that are typically triggered by a user interacting with the user interface.

The framework has two types of events.

- **Component events** are handled by the component itself or a component that instantiates or contains the component.
- **Application events** are handled by all components that are listening to the event. These events are essentially a traditional publish-subscribe model.

Interfaces

Object-oriented languages, such as Java, support the concept of an interface that defines a set of method signatures. A class that implements the interface provides the method implementations. An interface in Java can't be instantiated directly, but a class that implements the interface can. Similarly, the Lightning Component framework supports the concept of interfaces that define a component's shape by defining its attributes.

Open Source Aura Framework

The Lightning Component framework is built on the open source Aura framework. The Aura framework enables you to build apps independent of your data in Salesforce.

The Aura framework is available at <https://github.com/forcedotcom/aura>. The open source Aura framework has features and components that aren't yet available in the Lightning Component framework. We're working to surface more of these features and components for Salesforce developers.

SEE ALSO:

[Supported Lightning Components, Interfaces, and Events for Communities](#)

[Lightning Component Developer Guide: What is the Lightning Component Framework?](#)

[Lightning Component Developer Guide: Why Use the Lightning Component Framework?](#)

Which Lightning Template Do I Use?

All Lightning templates for communities support custom Lightning components, but to completely reconfigure your community, we recommend using the Customer Service (Napili) template.

Customer Service (Napili)

The Customer Service (Napili) template is the most flexible and full-featured template. Users can post questions to the community, search for and view articles, collaborate, and contact support agents by creating cases. This template is also the only one that supports the creation of Lightning Bolt solutions and custom theme layout components, which let you completely transform the look and feel of your community. Because all Lightning Bolt solutions are based on the Customer Service (Napili) template, the customizations described in this guide also apply to Bolt solutions.

Partner Central

The Partner Central template is a responsive template designed for channel or third-party sales. You can recruit, build, and grow your partner network to drive channel sales and marketing together in a branded online space. You can configure lead distribution, deal registration, and marketing campaigns, or share training materials and sales collateral in a central space, and use reports to track your pipeline. This template doesn't support custom theme layout components or the creation of Lightning Bolt solutions.

Koa and Kokua

The Koa and Kokua templates are starting a phased retirement so from Summer '17, you can no longer use these templates to create communities. Salesforce still supports existing communities that were built using Koa and Kokua, but we recommend that you work with Salesforce Support to migrate your existing Koa and Kokua communities.

SEE ALSO:

[Salesforce Help: Which Community Template Should I Use?](#)

CHAPTER 2 Develop Lightning Communities: The Basics

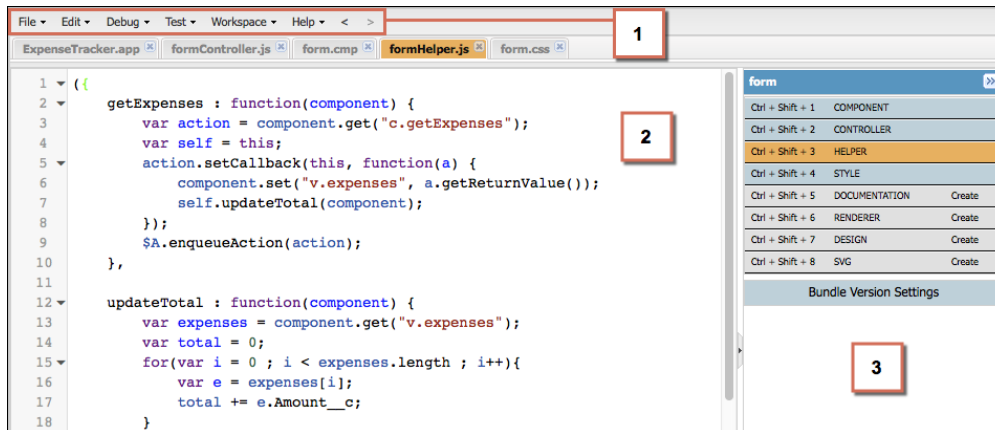
In this chapter ...

- Using the Developer Console
- Configure Drag-and-Drop Components for Community Builder
- Exposing Component Attributes in Community Builder
- Tips and Considerations for Configuring Components for Community Builder
- Supported Lightning Components, Interfaces, and Events for Communities

Learn about the Developer Console development tool, how to create a basic drag-and-drop Lightning component, and tips to consider along the way.

Using the Developer Console

The Developer Console provides tools for developing your components and applications.



The Developer Console enables you to perform these functions.

- Use the menu bar (1) to create or open these Lightning resources.
 - Application
 - Component
 - Interface
 - Event
 - Tokens
- Use the workspace (2) to work on your Lightning resources.
- Use the sidebar (3) to create or open client-side resources that are part of a specific component bundle.
 - Controller
 - Helper
 - Style
 - Documentation
 - Renderer
 - Design
 - SVG

For more information on the Developer Console, see [The Developer Console User Interface](#).

SEE ALSO:

[Salesforce Help: Open the Developer Console](#)

[Lightning Component Developer Guide: Create Lightning Components in the Developer Console](#)

[Lightning Component Developer Guide: Component Bundles](#)

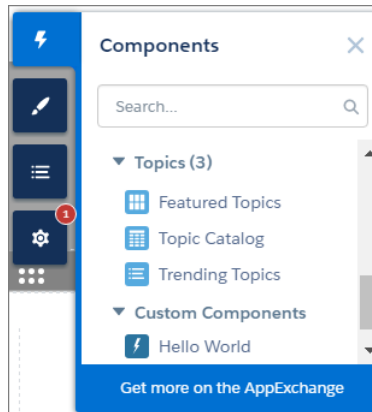
Configure Drag-and-Drop Components for Community Builder

Before you can use a custom Lightning component in Community Builder, there a few configuration steps to take.

1. Add an Interface to Your Component

To appear as a drag-and-drop component in Community Builder, a component must implement the `forceCommunity:availableForAllPageTypes` interface.

After you create the Lightning component, it appears in the Components panel of all Lightning communities in your org.



Here's the sample code for a simple "Hello World" component. A component must be named `componentName.cmp`.

Note: To make a resource, such as a component, usable outside of your own org, mark it with `access="global"`. For example, use `access="global"` if you want a component to be usable in an installed package or by a Community Builder user in another org.

```
<aura:component implements="forceCommunity:availableForAllPageTypes" access="global">
  <aura:attribute name="greeting" type="String" default="Hello" access="global" />
  <aura:attribute name="subject" type="String" default="World" access="global" />

  <div>{!v.greeting}, {!v.subject}</div>
</aura:component>
```

Warning: When you add custom components to your community, they can bypass the object- and field-level security (FLS) you set for the guest user profile. Lightning components don't automatically enforce [CRUD and FLS](#) when referencing objects or retrieving the objects from an Apex controller. This means that the framework continues to display records and fields for which users don't have CRUD permissions and FLS visibility. You *must* manually enforce CRUD and FLS in your Apex controllers.

2. Add a Design Resource to Your Component Bundle

A [design resource](#) controls which component attributes are exposed in Community Builder. The design resource lives in the same folder as your `.cmp` resource, and describes the design-time behavior of the Lightning component—information that visual tools need to display the component in a page or app.

For example, to set a default value for an attribute, or make a Lightning component attribute available for administrators to edit in Community Builder, your component bundle needs a design resource.


Here's the design resource that goes in the bundle with the "Hello World" component. A design resource must be named `componentName.design`.

```
<design:component label="Hello World">
  <design:attribute name="greeting" label="Greeting" />
</design:component>
```

```
<design:attribute name="subject" label="Subject" description="Name of the person you
want to greet" />
</design:component>
```

Optional. Add an SVG Resource to Your Component Bundle

To define a custom icon for your component, add an SVG resource to the component bundle. The icon appears next to the component in the Community Builder Components panel.

If you don't include an SVG resource, the system uses a default icon ().

Here's a simple red circle SVG resource to go with the "Hello World" component. An SVG resource must be named *componentName.svg*.

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
  width="400" height="400">
  <circle cx="100" cy="100" r="50" stroke="black"
    stroke-width="5" fill="red" />
</svg>
```

Optional. Add a CSS Resource to Your Component Bundle

To style your custom component, add a CSS resource to the component bundle.

Here's the CSS for a simple class to go with the "Hello World" component. A CSS resource must be named *componentName.css*.

```
.THIS .greeting {
  color: #ffe4e1;
  font-size: 20px;
}
```

After you create the class, apply it to your component.

```
<aura:component implements="forceCommunity:availableForAllPageTypes" access="global">
  <aura:attribute name="greeting" type="String" default="Hello" access="global" />
  <aura:attribute name="subject" type="String" default="World" access="global" />

  <div class="greeting">{!v.greeting}, {!v.subject}!</div>
</aura:component>
```

SEE ALSO:

[Lightning Component Developer Guide: Browser Support Considerations for Lightning Components](#)

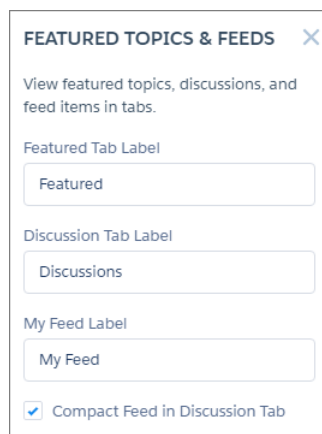
Exposing Component Attributes in Community Builder

You use a design resource to control which attributes are exposed in Community Builder. A design resource lives in the same folder as your component. It describes the design-time behavior of the Lightning component—information that visual tools need to display the component in a page or app.

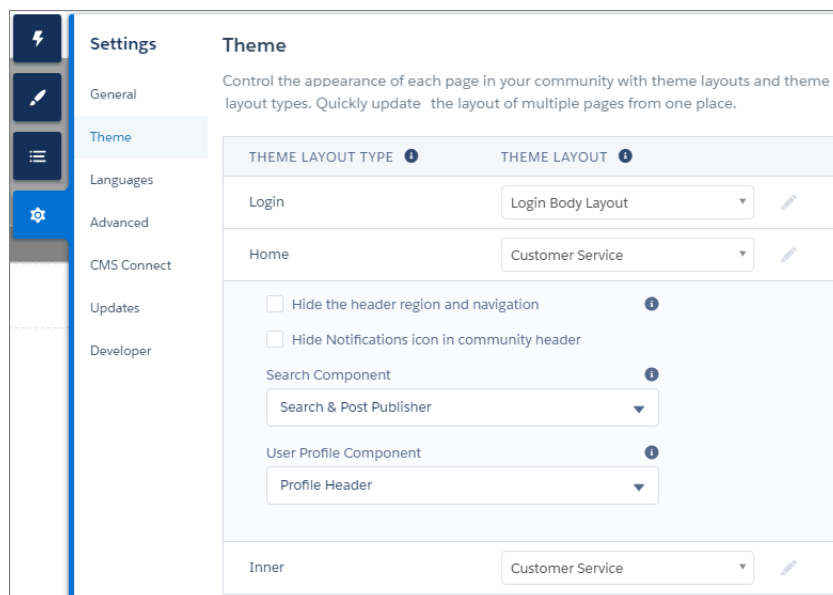
To make a Lightning component attribute available for administrators to edit in Community Builder, add a `design:attribute` node for the attribute in the design resource. When you mark an attribute as required, it automatically appears in Community Builder, unless it has a default value assigned to it.

You must specify required attributes with default values and attributes not marked as required in the component definition in the design resource to make them appear for users. A design resource supports attributes only of type `int`, `string`, or `boolean`.

For drag-and-drop components, exposed attributes appear in the component’s properties panel.



For theme layout components, exposed attributes appear when the theme layout is selected in the **Settings > Theme** area.



SEE ALSO:

[Lightning Component Developer Guide: Lightning Component Bundle Design Resources](#)

Tips and Considerations for Configuring Components for Community Builder

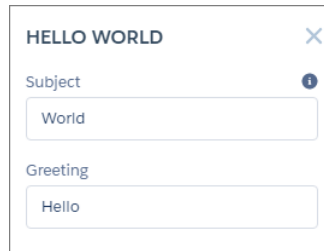
Keep these guidelines in mind when creating components and component bundles for Community Builder.

Components

- Give the component a friendly name using the `label` attribute in the design file element, such as `<design:component label="foo">`.
- Design your components to fill 100% of the width, including margins, of the region that they display in.
- Make sure that the component provides an appropriate placeholder behavior in declarative tools if it requires interaction.
- Never let a component display a blank box. Think of how other sites work. For example, Facebook displays an outline of the feed before the feed items come back from the server, which improves the user's perception of UI responsiveness.
- If the component depends on a fired event, give it a default state that displays before the event fires.
- Style components using [standard design tokens](#) to make them consistent with the Salesforce Design System.
- Keep in mind that [LockerService](#) is enforced for all Lightning components created in Summer '17 (API version 40.0) and later.

Attributes

- Use the design file to control which attributes are exposed to Community Builder.
- Make your attributes easy to use and understandable to an administrator. Don't expose SOQL queries, JSON objects, or Apex class names.
- Give required attributes default values to avoid a poor user experience. When a component that has required attributes with no default values is added to Community Builder, it appears invalid.
- Use basic supported types (`string`, `integer`, `boolean`) for exposed attributes.
- Specify a min and max for integer attributes in the `<design:attribute>` element to control the range of accepted values.
- Be aware that string attributes can provide a data source with a set of predefined values, allowing the attribute to expose its configuration as a picklist.
- Give attributes a label with a friendly display name.
- Include a description to explain the expected data and provide guidelines, such as the data format or expected range of values. Description text appears as a tooltip in the property panel.

A screenshot of a dialog box titled "HELLO WORLD" with a close button (X) in the top right corner. Below the title, there are two input fields. The first is labeled "Subject" and contains the text "World". The second is labeled "Greeting" and contains the text "Hello".

- To delete a design attribute for a component that implements the `forceCommunity:availableForAllPageTypes` interface, first remove the interface from the component before deleting the design attribute. Then reimplement the interface. If the component is referenced in a Lightning page, you must remove the component from the page before you can change it.

SEE ALSO:

[Lightning Component Developer Guide: Using the Salesforce Lightning Design System in Apps](#)

[Lightning Component Developer Guide: Styling with Design Tokens](#)

Supported Lightning Components, Interfaces, and Events for Communities

Not all Lightning components, interfaces, and events are supported for Communities. Some are available only for the Salesforce app or Lightning Experience. Check out what's available before customizing your community.

Interfaces

- `forceCommunity:availableForAllPageTypes`
- `forceCommunity:layout`
- `forceCommunity:profileMenuInterface`
- `forceCommunity:searchInterface`
- `forceCommunity:themeLayout`
- `forceHasRecordId`

Components

- `forceChatter:publisher`
- `forceCommunity:appLauncher`
- `forceCommunity:navigationMenuBase`
- `forceCommunity:notifications`
- `forceCommunity:routeLink`
- `forceCommunity:waveDashboard`

Events

- `forceCommunity:analyticsInteraction`

- `force:createRecord`
- `force:editRecord`
- `force:navigateToObject`
- `force:navigateToList`
- `force:navigateToRelatedList`
- `force:navigateToURL`
- `force:navigateToObjectHome`
- `lightning:openFiles`
- `force:refreshView`
- `forceCommunity:routeChange`
- `forceCommunity:setActiveLanguage`
- `force:showToast` (not available on login pages)

SEE ALSO:

[Lightning Component Developer Guide: Component Reference](#)

[Lightning Component Developer Guide: Interface Reference](#)

[Lightning Component Developer Guide: Event Reference](#)

CHAPTER 3 Customize the Look and Feel of a Lightning Template

In this chapter ...

- [Update a Template with the Branding Panel](#)
- [Override Template Elements with Custom CSS](#)
- [Use Custom Fonts in Your Community](#)
- [Customize the Theme Layout of Your Template](#)
- [Create Custom Content Layout Components for Communities](#)
- [Configure Swappable Search and Profile Menu Components](#)
- [Standard Design Tokens for Communities](#)

You can control the appearance of a Lightning template in several ways, each of varying complexity and granularity.

Within Community Builder, you can modify styles that are specific to the template and therefore can't be shared between communities. The options in Community Builder are the simplest to use and don't require coding.

- [The Branding panel](#) updates the template with simple, point-and-click branding properties. This option is ideal for admins to use.
- [The CSS Editor](#) lets you create custom CSS that overrides the basic styles of template elements. This option is suitable if you're familiar with CSS and want to make only minor modifications to some out-of-the-box components or template elements.

However, to completely customize the appearance of a template, you need to build your own components.

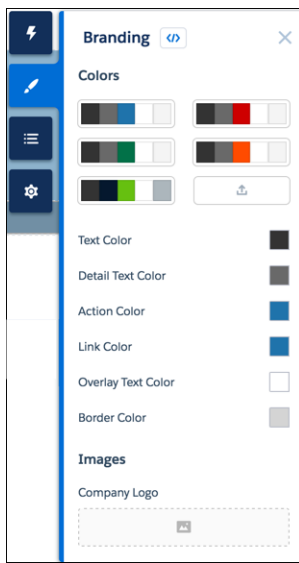
- Custom Lightning components encapsulate a CSS resource as part of the component bundle, making the components reusable across communities.
- [Content layout components](#) define the content regions of your page and contain components.
- [Theme layout components](#) let you customize the structural layout of the template, such as the header and footer, and override its default styles.

Update a Template with the Branding Panel

Within Community Builder, the simplest way to change the look of a template is with the Branding panel. Administrators can quickly style an entire community using branding properties to apply colors, specify fonts, and add a logo.

The properties set in the Branding panel apply to the pages in a template and most off-the-shelf components. However, a few components don't respond to branding property changes, which is a limitation of solely using the Branding panel.

The Branding panel's properties also apply to custom Lightning components that use [standard design tokens](#) to control their appearance.



Note: To unify the branding properties of the login pages with the rest of the pages of your community, update your template in **Settings > Update**. Otherwise, you have to brand the login pages separately.

SEE ALSO:

[Salesforce Help: Update Your Community's Template](#)

[Salesforce Help: Brand Your Community with Community Builder](#)

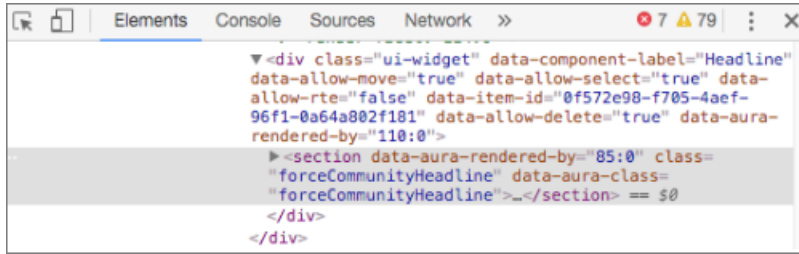
Override Template Elements with Custom CSS

Use the CSS Editor in Community Builder to add custom CSS that overrides the default template and Branding panel styles. You can also use it to make minor changes to the appearance of out-of-the-box components, such as padding adjustments. However, use custom CSS sparingly because future releases of template components might not support your CSS customizations.

Note: For large customizations, use the CSS resource in custom Lightning components and custom theme layout components instead of custom CSS. If you use global overrides, always test your community in sandbox when it's updated each release.

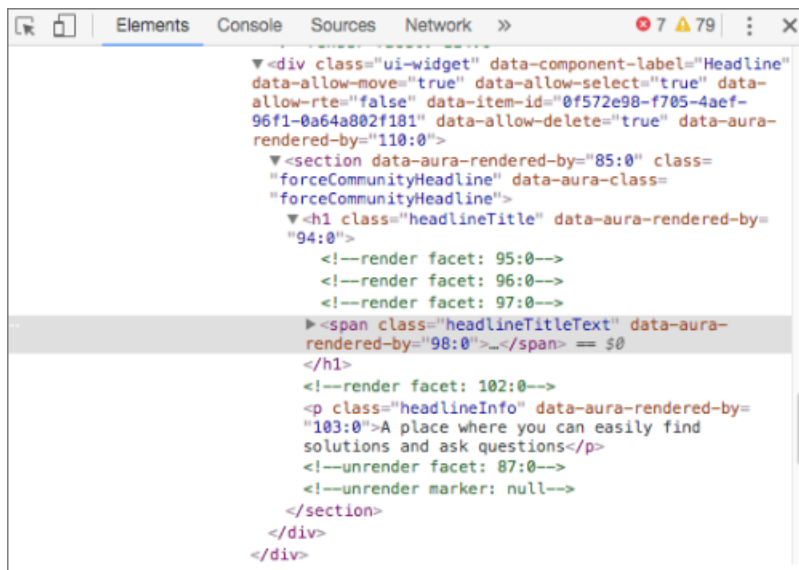
To make minor CSS modifications to a template item, use Chrome DevTools to inspect the page and discover the item's fully qualified name and CSS class. Then use the information to override the item's standard CSS with your custom CSS. To learn more about inspecting and editing pages and styles, refer to the [Google Chrome's DevTools](#) website.

The easiest way to inspect a component is to view the page in Preview mode. This example inspects the Headline component to locate the component's fully qualified name—`forceCommunityHeadline`.



Note: If a top-level CSS class isn't defined for a component, this option doesn't appear, which means that you can't reliably target the component.

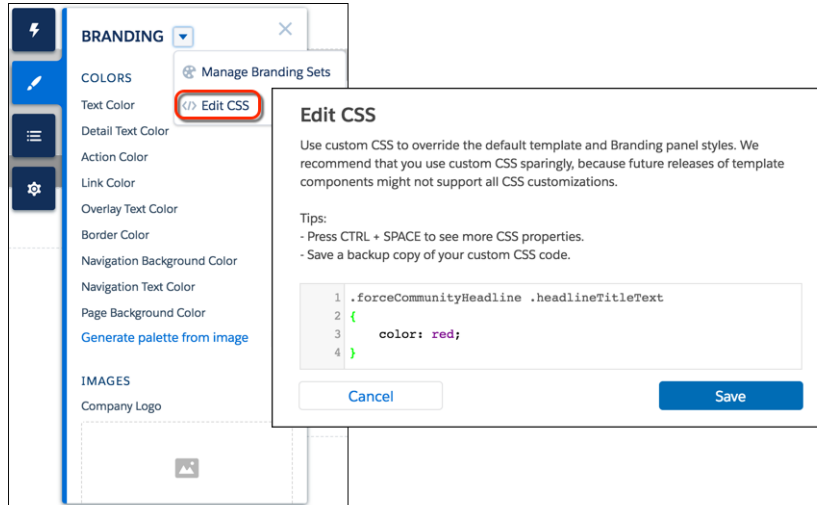
Then find the element that you want to style—for example, `headlineTitleText`. If the element doesn't have a class name, you must write a specific selector targeting the element.



With that information, you can create a custom style to override the default title color.

```
.forceCommunityHeadline .headlineTitleText
{
    color: red;
}
```

And then add it to the CSS Editor.



Similarly, you could use custom CSS to hide the component entirely.

```
.forceCommunityHeadline
{
  display: none;
}
```

Tip: You can link to a CSS style sheet as either a static or external resource in the head markup in **Settings > Advanced**. However, because global value providers aren't supported in the head markup or in CSS overrides, you can't use `$resource` to reference static resources. Instead, use a relative URL using the syntax `/sfsites/c/resource/resource_name`.

For example, if you upload an image as a static resource called Headline, reference it in the CSS Editor as follows:

```
.forceCommunityHeadline
{
  background-image: url('/sfsites/c/resource/headline')
}
```

Head markup is also useful for adding favicon icons, SEO meta tags, and other items. However, be aware of [future stricter CSP restrictions](#) that could affect your code.

Migrate CSS Overrides

Many CSS selectors changed with the Customer Service (Napili) update. If your communities use custom CSS overrides to the default template and Branding panel styles, you must update them to use the new selectors. Two new branding properties, Navigation Background Color and Navigation Text Color, reduce the need for these overrides. If you plan to continue using custom CSS overrides, migrate them forward after you update the template.

SEE ALSO:

[Salesforce Help: Static Resources](#)

[Salesforce Help: Add Markup to the Page <head> to Customize Your Community](#)

Migrate CSS Overrides

Many CSS selectors changed with the Customer Service (Napili) update. If your communities use custom CSS overrides to the default template and Branding panel styles, you must update them to use the new selectors. Two new branding properties, Navigation Background Color and Navigation Text Color, reduce the need for these overrides. If you plan to continue using custom CSS overrides, migrate them forward after you update the template.

This topic identifies selector changes.



Note:

- Use custom CSS sparingly because template updates might not support your customizations.
- Custom CSS is now shared across all your community pages. If you used custom CSS for Login pages, copy it and close the CSS editor. Then navigate to a non-Login page, reopen the editor, and add the custom CSS.

Full Navigation Menu

Previous Selector	New Selector
<pre>.forceCommunityNavigationMenu .forceCommunityNavigationMenu #navigationMenu .forceCommunityNavigationMenu .navigationMenu .forceCommunityNavigationMenu .navigationMenuWrapper</pre>	<pre>.comm-navigation</pre>

Mobile Menu Curtain

Previous Selector	New Selector
<pre>.forceCommunityNavigationMenu .navigationMenuWrapperCurtain</pre>	<pre>.comm-navigation nav.slds-is-fixed</pre>

Home Menu Item

Previous Selector	New Selector
<pre>.forceCommunityNavigationMenu .homeLink .forceCommunityNavigationMenu .homeButton</pre>	<pre>.comm-navigation .slds-list__item a[data-type="home"]</pre>

Mobile Menu Toggle Button

Previous Selector	New Selector
<code>.forceCommunityNavigationMenu .toggleNav</code>	<code>.siteforceServiceBody .cHeaderPanel .cAltToggleNav</code>

Top-Level Menu Items

Includes submenu triggers.

Previous Selector	New Selector
<code>.forceCommunityNavigationMenu .menuItem</code> <code>.forceCommunityGlobalNavigation .navigationMenuNode</code>	<code>.comm-navigation .comm-navigation__list > .slds-list__item</code>

Current Top-Level Menu Item

Previous Selector	New Selector
<code>.forceCommunityNavigationMenu .current</code> <code>.forceCommunityGlobalNavigation .menuItem.current</code>	<code>.comm-navigation .comm-navigation__list > .slds-list__item > .slds-is-active</code>

Top-Level Menu Item Links

Previous Selector	New Selector
<code>.forceCommunityNavigationMenu .menuItemLink</code> <code>.forceCommunityNavigationMenu a.menuItemLink</code> <code>.forceCommunityNavigationMenu .menuItem .menuItemLink</code> <code>.forceCommunityNavigationMenu .menuItem a</code> <code>.forceCommunityNavigationMenu .menuItem a.menuItemLink</code>	<code>.comm-navigation .comm-navigation__list > .slds-list__item > a</code> <code>.comm-navigation .comm-navigation__list > .slds-list__item > button</code>

Submenu Items

Previous Selector	New Selector
<code>.forceCommunityNavigationMenu .subMenuItem</code>	<code>.comm-navigation .slds-list_vertical.slds-is-nested .slds-list__item</code>

Current/Active Submenu Item

Previous Selector	New Selector
<code>.forceCommunityNavigationMenu .subMenuItem.current</code>	<code>.comm-navigation .slds-list_vertical.slds-is-nested .slds-list__item .slds-is-active</code>

Submenu Trigger Link

Previous Selector	New Selector
<code>.forceCommunityNavigationMenu .triggerLink .forceCommunityNavigationMenu .triggerLabel</code>	<code>.comm-navigation .slds-list__item button:enabled</code>

Submenu Trigger Link Icon

Previous Selector	New Selector
<code>.forceCommunityNavigationMenu .triggerLink .forceIcon</code>	<code>.comm-navigation .slds-list__item button:enabled .slds-icon_container</code>

Menu Items

Includes top-level and submenu items.

Previous Selector	New Selector
<code>.forceCommunityNavigationMenu .navigationMenu li</code>	<code>.comm-navigation .slds-list__item</code>

Menu Item Links

Includes top-level and submenu items.

Previous Selector	New Selector
<pre>.forceCommunityNavigationMenu a .forceCommunityNavigationMenu a.menuItemLink</pre>	<pre>.comm-navigation .slds-list__item a .comm-navigation .slds-list__item button</pre>

Submenus

Previous Selector	New Selector
<pre>.forceCommunityNavigationMenu .subMenu</pre>	<pre>.comm-navigation .slds-list_vertical.slds-is-nested</pre>

Submenu Items

Previous Selector	New Selector
<pre>.forceCommunityNavigationMenu .subMenuItem</pre>	<pre>.comm-navigation .slds-list_vertical.slds-is-nested .slds-list__item</pre>

Submenu Item Links

Previous Selector	New Selector
<pre>.forceCommunityNavigationMenu .subMenuItem a .forceCommunityNavigationMenu .subMenu a</pre>	<pre>.comm-navigation .slds-list_vertical.slds-is-nested .slds-list__item a</pre>

SEE ALSO:

[Override Template Elements with Custom CSS](#)

Use Custom Fonts in Your Community

Upload custom fonts as static resources and use them for primary and header fonts throughout your community. If you have more than one font file to upload, use a .zip file.

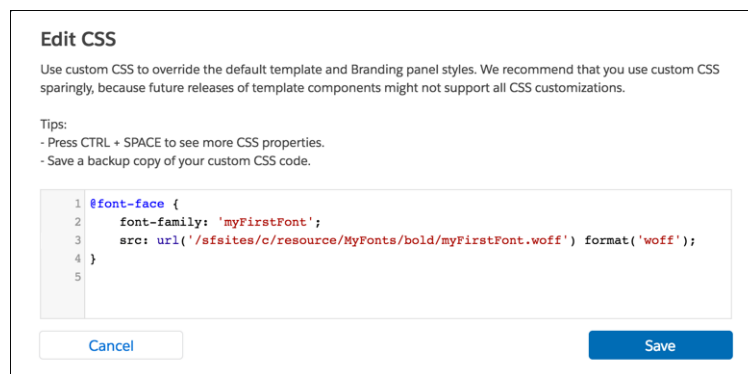
1. In Setup, enter *Static Resources* in the Quick Find box and click **Static Resources**.
2. Click **New**, upload the file, and give the resource a name. Keep a note of the static resource name.
3. In the CSS Editor in Community Builder, use the `@font-face` CSS rule to reference the uploaded font.

To reference a single font file, use the syntax `/sfsites/c/resource/resource_name`. For example, if you upload a file called `myFirstFont.woff` and name the resource `MyFonts`, the URL is `/sfsites/c/resource/MyFonts`.

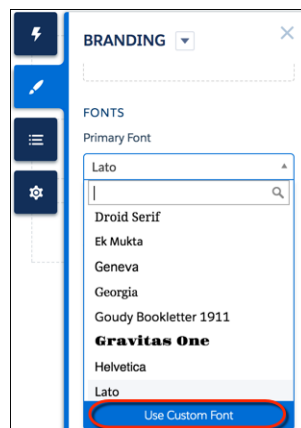
To reference a file in a `.zip` file, include the folder structure but omit the `.zip` file name. Use the syntax `/sfsites/c/resource/resource_name/font_folder/font_file`. So if you upload `fonts.zip`, which contains `bold/myFirstFont.woff`, and you name the resource `MyFonts`, the URL is `/sfsites/c/resource/MyFonts/bold/myFirstFont.woff`.

For example:

```
/* example font */
@font-face {
  font-family: 'myFirstFont';
  src: url('/sfsites/c/resource/MyFonts/bold/myFirstFont.woff') format('woff');
}
```



4. In the Branding panel, under Fonts, click **Use Custom Font** and add the font family name.



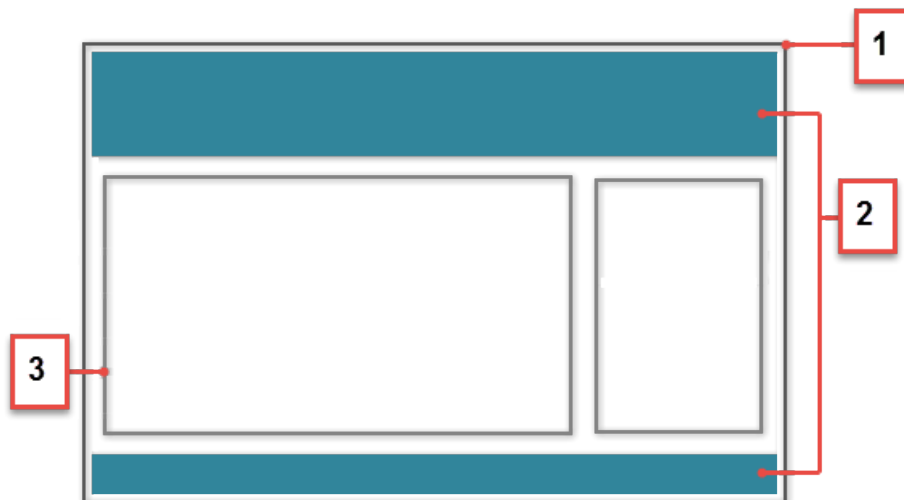
Use Custom Font

To use a custom font throughout your community, upload the custom font and define it in the CSS Editor. Then add the font family name here. [Learn More](#)

Customize the Theme Layout of Your Template

To put your own stamp on a template and transform its appearance, build a custom theme layout component. You can customize the template's structural layout, such as the header and footer, and override its default styles.

A theme layout is the top-level layout for the template pages (1) in your community. It includes the common header and footer (2), and often includes navigation, search, and the user profile menu. In contrast, the content layout (3) defines the content regions of your pages, such as a two-column layout.



SEE ALSO:

[Example: Build a Condensed Theme Layout Component](#)

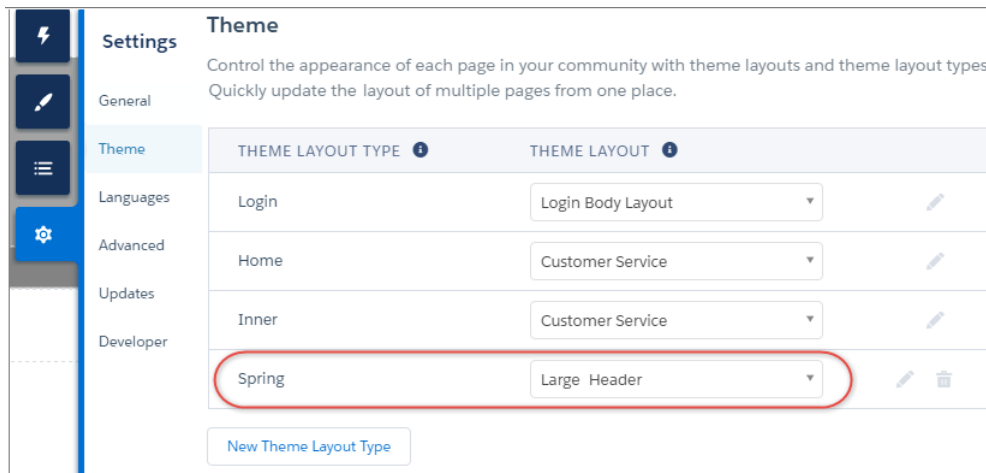
How Do Custom Theme Layouts Work?

To understand how a theme layout works, let's look at things from the Community Builder perspective. In Community Builder, a theme layout combines with theme layout types to give you granular control of the appearance and structure of each page in your community. You can customize the layout's header and footer to match your company's style, configure theme properties, or use a custom search bar and user profile menu. You then use theme layout types to apply a theme layout to individual pages and quickly change layouts from one central location.

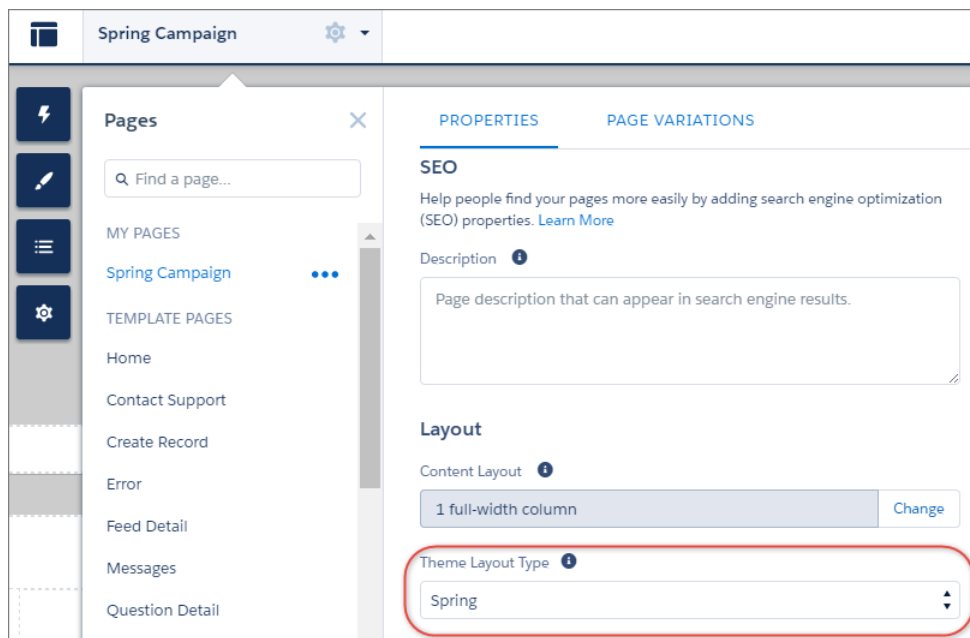
A theme layout type categorizes the pages in your community that share the same theme layout. You can assign a theme layout to an existing type or create custom types. Then you apply the theme layout type—and thereby the theme layout—in the page's properties. Customer Service (Napili) includes the following theme layouts and types.

- Inner applies the Customer Service theme layout to all pages, except the login pages.
- Login applies the Login Body Layout theme layout to the login pages.
- Home isn't applied to any pages. However, the Home layout type is ideal for applying a separate theme layout when you want your home page to look different from the inner pages in your community.

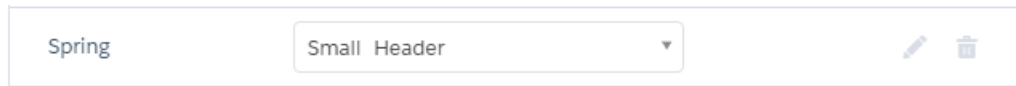
Example: Let's say you create three pages for your upcoming Spring campaign. Using the `forceCommunity:themeLayout` interface, you create a custom Large Header theme layout in the Developer Console. In the **Settings > Theme** area, you add a custom theme layout type called Spring to categorize the campaign pages and you assign the Large Header layout to it.



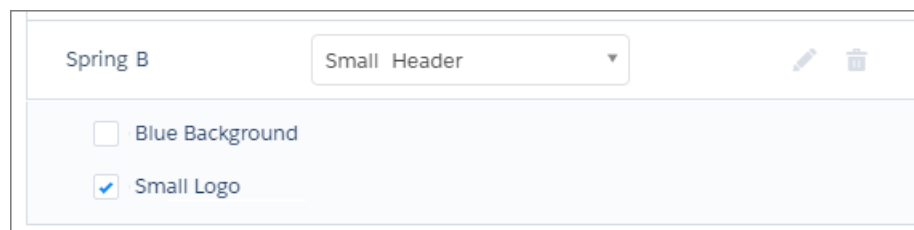
Next, you apply the Spring theme layout type in each page's properties, which instantly applies the Large Header layout to each page.



Everything looks rosy until the VP of marketing decides that the header takes up too much room. That's an easy fix, because you don't have to update the properties of each page to change the theme layout. Instead, with one click in the Theme area, you can switch Spring to the Small Header layout and instantly update all three pages.



Example: Now let's say you have two custom properties—Blue Background and Small Logo—in the Small Header layout, which you've enabled and applied to all your campaign pages. However, for one page, you want to apply only the Small Logo property. In this case, you can create a theme layout type called Spring B, assign the Small Header layout to it, and enable Small Logo. Then apply the Spring B layout type to the page.



Theme layout types make it easy to reuse the same theme layout in different ways while maintaining as much granular control as you need.

SEE ALSO:

[Example: Build a Condensed Theme Layout Component](#)

Configure a Custom Theme Layout Component

Let's look at how to create a custom theme layout in the Developer Console to transform the appearance and overall structure of the pages in the Customer Service (Napili) template.

1. Add an Interface to Your Theme Layout Component

A theme layout component must implement the `forceCommunity:themeLayout` interface to appear in Community Builder in the **Settings > Theme** area.

Explicitly declare `{!v.body}` in your code to ensure that your theme layout includes the content layout. Add `{!v.body}` wherever you want the page's contents to appear within the theme layout.

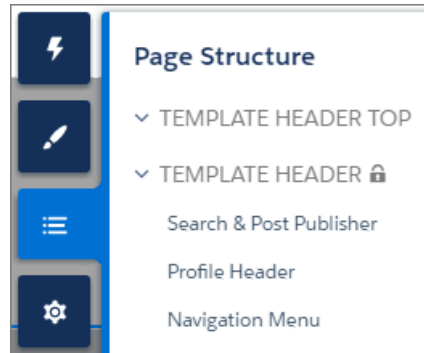
Add attributes declared as `Aura.Component[]` to include regions in the theme layout, which contain the page's components. You can add components to the regions in your markup or leave regions open for users to drag-and-drop components into. Attributes declared as `Aura.Component[]` and included in your markup are rendered as open regions in the theme layout that users can add components to. For example:

```
<aura:component implements="forceCommunity:themeLayout">
<aura:attribute name="myRegion" type="Aura.Component[]"/>
```

```
{!v.body}
</aura:component>
```

In Customer Service (Napili), the Template Header consists of these locked regions:

- `search`, which contains the Search Publisher component
- `profileMenu`, which contains the Profile Header component
- `navBar`, which contains the Navigation Menu component



To create a custom theme layout that reuses the existing components in the Template Header region, declare `search`, `profileMenu`, or `navBar` as the attribute name value, as appropriate. For example:

```
<aura:attribute name="navBar" type="Aura.Component[]" required="false" />
```

Tip: If you create a [swappable custom profile menu](#) or a [search component](#), declaring the `search` or `profileMenu` attribute name value also lets users select the custom component when using your theme layout in Community Builder.

Add the regions to your markup to define where to display them in the theme layout's body.

Here's the sample code for a simple theme layout.

```
<aura:component implements="forceCommunity:themeLayout" access="global" description="Sample Custom Theme Layout">
  <aura:attribute name="search" type="Aura.Component[]" required="false"/>
  <aura:attribute name="profileMenu" type="Aura.Component[]" required="false"/>
  <aura:attribute name="navBar" type="Aura.Component[]" required="false"/>
  <aura:attribute name="newHeader" type="Aura.Component[]" required="false"/>
  <div>
    <div class="searchRegion">
      {!v.search}
    </div>
    <div class="profileMenuRegion">
      {!v.profileMenu}
    </div>
    <div class="navigation">
      {!v.navBar}
    </div>
    <div class="newHeader">
      {!v.newHeader}
    </div>
    <div class="mainContentArea">
```

```

        {!v.body}
      </div>
    </div>
  </aura:component>

```

2. Add a Design Resource to Include Theme Properties

You can expose theme layout properties in Community Builder by adding a design resource to your bundle.

First, implement the properties in the component.

```

<aura:component implements="forceCommunity:themeLayout" access="global" description="Small
Header">
  <aura:attribute name="blueBackground" type="Boolean" default="false"/>
  <aura:attribute name="smallLogo" type="Boolean" default="false" />
  ...

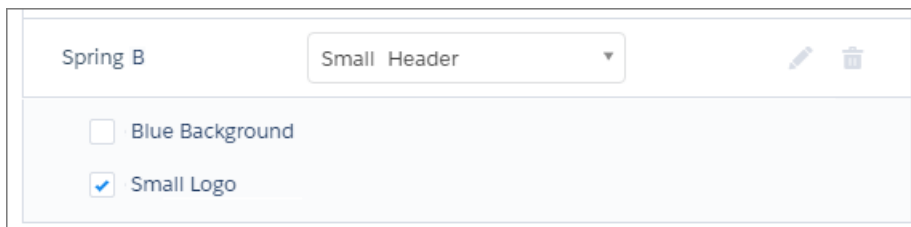
```

Define the theme properties in the design resource to expose the properties in the UI. This example adds a label for the Small Header theme layout along with two checkboxes.

```

<design:component label="Small Header">
  <design:attribute name="blueBackground" label="Blue Background"/>
  <design:attribute name="smallLogo" label="Small Logo"/>
</design:component>

```



3. Add a CSS Resource to Avoid Overlapping Issues

Add a CSS resource to your bundle to style the theme layout as needed, ideally using [standard design tokens](#).

To avoid overlapping issues with positioned elements, such as dialog boxes or hovers:

- Apply CSS styles.

```

.THIS {
  position: relative;
  z-index: 1;
}


```

- Wrap the elements in your custom theme layout in a `div` tag.

```

<div class="mainContentArea">
  {!v.body}
</div>

```

 **Note:** The theme layout controls the styling of anything within it, so it can add styles such as drop-shadows to regions or components. For custom theme layouts, SLDS is loaded by default.

SEE ALSO:

[Example: Build a Condensed Theme Layout Component](#)

Create Custom Content Layout Components for Communities

Community Builder includes several ready-to-use layouts that define the content regions of your page, such as a two-column layout with a 2:1 ratio. However, if you need a layout that's customized for your community, create a custom content layout component to use when building new pages in Community Builder. You can also update the content layout of the default pages that come with your community template.

When you create a custom content layout component in the Developer Console, it appears in Community Builder in the New Page and the Change Layout dialog boxes.


1. Add a New Interface to Your Content Layout Component

To appear in the New Page and the Change Layout dialog boxes in Community Builder, a content layout component must implement the `forceCommunity:layout` interface.

Here's the sample code for a simple two-column content layout.

```
<aura:component implements="forceCommunity:layout" description="Custom Content Layout"
access="global">
  <aura:attribute name="column1" type="Aura.Component[]" required="false"></aura:attribute>
  <aura:attribute name="column2" type="Aura.Component[]" required="false"></aura:attribute>

  <div class="container">
    <div class="contentPanel">
      <div class="left">
        {!v.column1}
      </div>
      <div class="right">
        {!v.column2}
      </div>
    </div>
  </div>
</aura:component>
```

 **Note:** Mark your resources, such as a component, with `access="global"` to make the resource usable outside of your own org. For example, if you want a component to be usable in an installed package or by a Lightning App Builder user or a Community Builder user in another org.

2. Add a CSS Resource to Your Component Bundle

Next, add a CSS resource to style the content layout as needed.

Here's the sample CSS for our simple two-column content layout.

```
.THIS .contentPanel:before,  
.THIS .contentPanel:after {  
  content: " ";  
  display: table;  
}  
.THIS .contentPanel:after {  
  clear: both;  
}  
.THIS .left {  
  float: left;  
  width: 50%;  
}  
.THIS .right {  
  float: right;  
  width: 50%;  
}
```

CSS resources must be named `componentName.css`.

3. Optional: Add an SVG Resource to Your Component Bundle

You can include an SVG resource in your component bundle to define a custom icon for the content layout component when it appears in the Community Builder.

The recommended image size for a content layout component in Community Builder is 170px by 170px. However, if the image has different dimensions, Community Builder scales the image to fit.

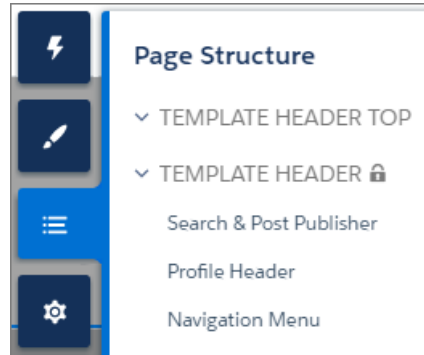
SVG resources must be named `componentName.svg`.

Configure Swappable Search and Profile Menu Components

Create custom components to replace the Customer Service (Napili) template's standard Profile Header and Search & Post Publisher components in Community Builder.

In Customer Service (Napili), the Template Header consists of these locked regions:

- `search`, which contains the Search Publisher component
- `profileMenu`, which contains the Profile Header component
- `navBar`, which contains the Navigation Menu component



These designated region names let you easily:

- Swap search and profile components in the default Customer Service theme layout or a custom theme layout.
- Swap theme layout components while persisting existing customizations, such as the selected search component.

When a component implements the correct interface—`forceCommunity:searchInterface` or `forceCommunity:profileMenuInterface`, in this case—it's identified as a candidate for these regions. They therefore appear as swappable components in a theme layout, such as the default Customer Service theme layout, which declares `search` or `profileMenu` as an attribute name value.

```
<aura:attribute name="search" type="Aura.Component[]" required="false" />
```

`forceCommunity:profileMenuInterface`

Add the `forceCommunity:profileMenuInterface` interface to a Lightning component to allow it to be used as a custom profile menu component for the Customer Service (Napili) community template. After you create a custom profile menu component, admins can select it in Community Builder in **Settings > Theme** to replace the template's standard Profile Header component.

This code is for a simple profile menu component.

```
<aura:component implements="forceCommunity:profileMenuInterface" access="global">
  <aura:attribute name="options" type="String[]" default="Option 1, Option 2"/>
  <ui:menu >
    <ui:menuTriggerLink aura:id="trigger" label="Profile Menu"/>
    <ui:menuList class="actionMenu" aura:id="actionMenu">
      <aura:iteration items="{!v.options}" var="itemLabel">
        <ui:actionMenuItem label="{!itemLabel}" click="{!c.handleClick}"/>
      </aura:iteration>
    </ui:menuList>
  </ui:menu>
</aura:component>
```

`forceCommunity:searchInterface`

Add the `forceCommunity:searchInterface` interface to a Lightning component to allow it to be used as a custom search component for the Customer Service (Napili) community template. After you create a custom search component, admins can select it in Community Builder in **Settings > Theme** to replace the template's standard Search & Post Publisher component.

This code is for a simple search component.

```
<aura:component implements="forceCommunity:searchInterface" access="global">
  <div class="search">
    <div class="search-wrapper">
      <form class="search-form">
        <div class="search-input-wrapper">
          <input class="search-input" type="text" placeholder="My Search"/>
        </div>
        <input type="hidden" name="language" value="en" />
      </form>
    </div>
  </div>
</aura:component>
```

SEE ALSO:

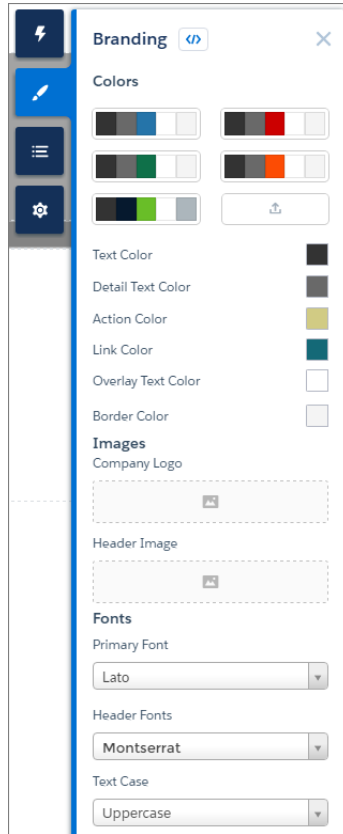
[Step 5: Build a Custom Search Component](#)

Lightning Component Developer Guide: forceCommunity:navigationMenuBase

Standard Design Tokens for Communities

Salesforce exposes a set of base tokens that you can access in your component style resources. You can use these standard tokens to mimic the look-and-feel of the Salesforce Lightning Design System (SLDS) in your own custom components. As the SLDS evolves, components that are styled using the standard design tokens evolve along with it. Use a subset of the standard design tokens to make your components compatible with the Branding panel in Community Builder.

With the Branding panel, administrators can quickly style an entire community using branding properties. Each property in the Branding panel maps to one or more standard design tokens. When an administrator updates a property in the Branding panel, the system updates the Lightning components that use the tokens associated with that branding property.



Available Tokens for Communities

For Communities using the Customer Service (Napili) template, the following standard tokens are available when extending from `force:base`.

Important: The standard token values evolve along with SLDS. Available tokens and their values can change without notice.


These Branding panel properties...	...map to these standard design tokens
Text Color	<code>colorTextDefault</code>
Detail Text Color	<ul style="list-style-type: none"> <code>colorTextLabel</code> <code>colorTextPlaceholder</code> <code>colorTextWeak</code>
Action Color	<ul style="list-style-type: none"> <code>colorBackgroundButtonBrand</code> <code>colorBackgroundHighlight</code> <code>colorBorderBrand</code> <code>colorBorderButtonBrand</code> <code>colorBrand</code> <code>colorTextBrand</code>

These Branding panel properties...	...map to these standard design tokens
Link Color	<code>colorTextLink</code>
Overlay Text Color	<ul style="list-style-type: none"> • <code>colorTextButtonBrand</code> • <code>colorTextButtonBrandHover</code> • <code>colorTextInverse</code>
Border Color	<ul style="list-style-type: none"> • <code>colorBorder</code> • <code>colorBorderButtonDefault</code> • <code>colorBorderInput</code> • <code>colorBorderSeparatorAlt</code>
Primary Font	<code>fontFamily</code>
Text Case	<code>textTransform</code>

In addition, the following standard tokens are available for derived branding properties in the Customer Service (Napili) template. You can indirectly access derived branding properties when you update the properties in the Branding panel. For example, if you change the Action Color property in the Branding panel, the system recalculates the Action Color Darker value based on the new value.

These derived branding properties...	...map to these standard design tokens
Action Color Darker (Derived from Action Color)	<ul style="list-style-type: none"> • <code>colorBackgroundButtonBrandActive</code> • <code>colorBackgroundButtonBrandHover</code>
Hover Color (Derived from Action Color)	<ul style="list-style-type: none"> • <code>colorBackgroundButtonDefaultHover</code> • <code>colorBackgroundRowHover</code> • <code>colorBackgroundRowSelected</code> • <code>colorBackgroundShade</code>
Link Color Darker (Derived from Link Color)	<ul style="list-style-type: none"> • <code>colorTextLinkActive</code> • <code>colorTextLinkHover</code>

For a complete list of the design tokens available in the SLDS, see [Design Tokens](#) on the Lightning Design System site.

 **Note:** Several out-of-the-box components don't use standard design tokens. Therefore, if you use tokens when styling your theme layout, some components might not inherit the styles you define.

SEE ALSO:

- [Lightning Component Developer Guide: Styling with Design Tokens](#)
- [Lightning Component Developer Guide: Using the Salesforce Lightning Design System in Apps](#)

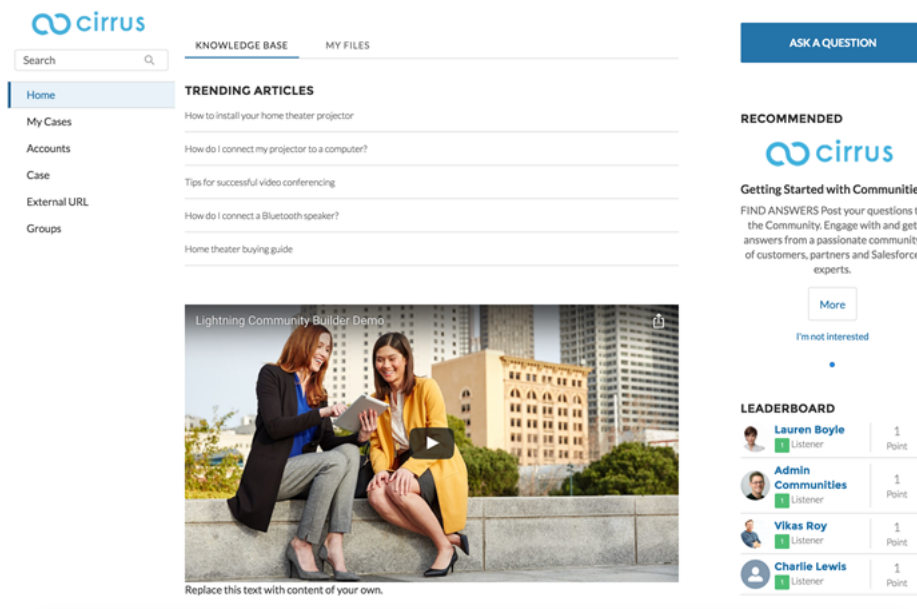
CHAPTER 4 Example: Build a Condensed Theme Layout Component

In this chapter ...

- Step 1: Create the Basic Theme Layout Structure
- Step 2: Define a Tokens Bundle
- Step 3: Add a Logo Component
- Step 4: Build a Vertical Navigation Menu
- Step 5: Build a Custom Search Component
- Step 6: Add Configuration Properties to the Theme Layout

Let's look at how to create a sample theme layout component for a home page, which uses a side navigation bar and a custom search component, and removes the header entirely.

Using the code samples in this section, you can create the skeleton for a custom theme layout.



Step 1: Create the Basic Theme Layout Structure

To create the basic structure of the theme layout, add attributes to define two regions—`search` and `sidebar`. Then add the attributes to the markup to define where the regions appear.

```
<aura:component implements="forceCommunity:themeLayout">
  <aura:attribute name="search" type="Aura.Component[]"/>
  <aura:attribute name="sidebarFooter" type="Aura.Component[]"/>

  {!v.search}
  {!v.sidebarFooter}
  {!v.body}

</aura:component>
```

Right now, all the regions flow vertically, so add some semantic structure using the [SLDS grid system](#).

```
<aura:component implements="forceCommunity:themeLayout">
  <aura:attribute name="search" type="Aura.Component[]"/>
  <aura:attribute name="sidebarFooter" type="Aura.Component[]"/>
  <div class="slds-grid slds-grid--align-center">
    <div class="slds-col">
      <div class="slds-grid slds-grid--vertical">
        <div class="slds-col">
          <!-- placeholder for logo -->
        </div>
        <div class="slds-col">
          {!v.search}
        </div>
        <div class="slds-col">
          <!-- placeholder for navigation -->
        </div>
        <div class="slds-col">
          {!v.sidebarFooter}
        </div>
      </div>
    </div>
    <div class="slds-col content">
      {!v.body}
    </div>
  </div>
</aura:component>
```

Add a design resource to the bundle to give the component a UI label.

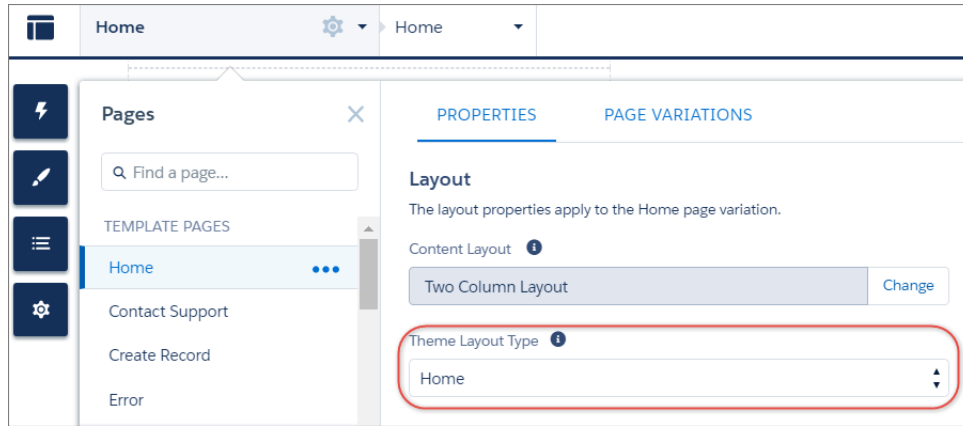
```
<design:component label="Condensed Theme Layout">

</design:component>
```

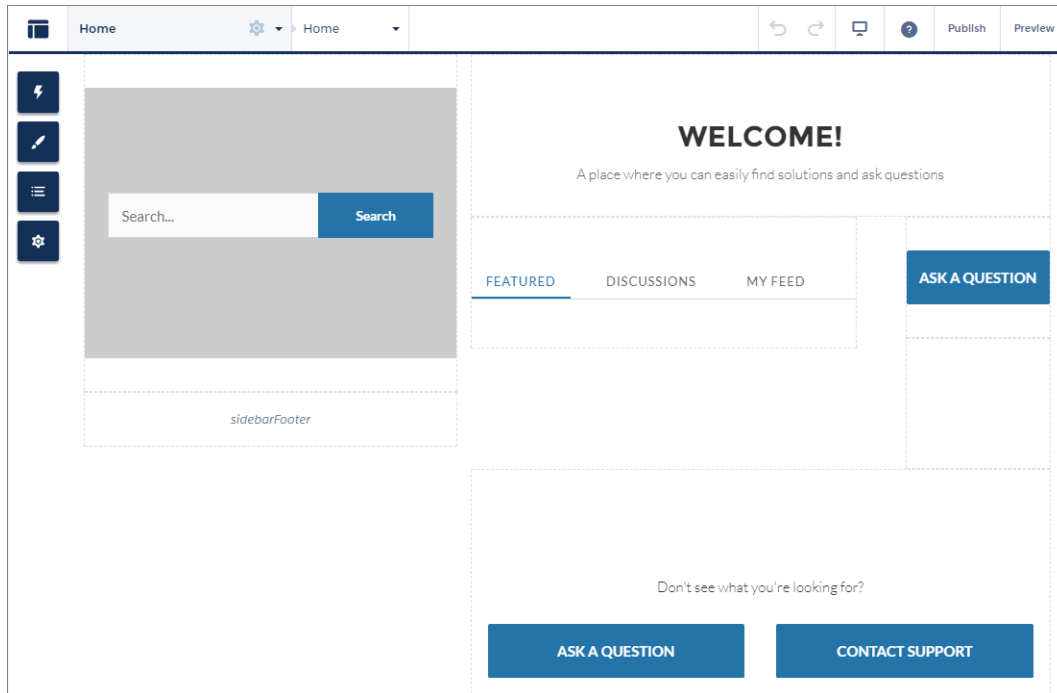
In Community Builder, you can see the theme layout's semantic hierarchy by selecting it for the Home theme layout type.

THEME LAYOUT TYPE ⓘ	THEME LAYOUT ⓘ
Login	Login Body Layout ▾
Home	Condensed Theme Layout ▾
Inner	Customer Service ▾

Open the Page Properties for the Home page. Select **Home** in the Theme Layout Type dropdown.



The page refreshes, and now you can see the new theme layout type in action. Let's inspect the layout of the page. You no longer have a header, which used to contain the navigation, search, profile menu, and logo. Some of those elements are being moved into the two left sidebar regions—`search` and `sidebarFooter`. However, until you create a swappable search component for the designated `search` region, the standard search component still appears.



SEE ALSO:

- [Configure a Custom Theme Layout Component](#)
- [How Do Custom Theme Layouts Work?](#)

Step 2: Define a Tokens Bundle

To enable your Lightning components to access branding tokens, define a tokens bundle in the same namespace. In the Developer Console, create a new tokens bundle with the name `defaultTokens`.

```
<aura:tokens extends="force:base">
</aura:tokens>
```

The tokens bundle extends `force:base`. By including `extends="force:base"` in your markup, you now have access to all the tokens exposed by SLDS and the branding values defined in the Branding panel in Community Builder.

SEE ALSO:

- [Standard Design Tokens for Communities](#)
- [Lightning Component Developer Guide: Standard Design Tokens—`force:base`](#)

Step 3: Add a Logo Component

Return to the theme layout component to add a company logo to it.

You can add a logo to the page in several ways. You can use:

- A custom component that displays a static resource—for example, `{!$Resource.MyJavascriptFile}`
- A custom component and fetch the path to the asset from the server
- An out-of-the-box component, such as the Rich Content Editor

For this example, let's use a custom component and reference a static resource called `cirruslogo`. In the theme layout component, add the following code to the first `slds-col` container with the logo placeholder comments.

```
<div class="logoContainer">
  
</div>
```

SEE ALSO:

[Lightning Component Developer Guide: \\$Resource](#)

[Salesforce Help: Static Resources](#)

Step 4: Build a Vertical Navigation Menu

To add a vertical navigation menu to the sidebar, create a new component named `verticalNav` that extends the abstract `forceCommunity:navigationMenuBase` component.

```
<aura:component extends="forceCommunity:navigationMenuBase">
</aura:component>
```

The component automatically has access to the navigation menu items defined in the community's Navigation Menu component. To see it working, create a quick unordered list of the navigation menu items.

```
<aura:component extends="forceCommunity:navigationMenuBase">
  <ul>
    <aura:iteration items="{!v.menuItems}" var="item">
      <li>{!item.label}</li>
    </aura:iteration>
  </ul>
</aura:component>
```

This simple unordered list iterates through an array of `menuItems`, which is defined in the extended abstract component, and outputs `` for each entry in the array.

To test the component, in the markup for `condensedThemeLayout`, add the component to the third column that has the placeholder comment for the navigation.

```
<c:verticalNav></c:verticalNav>
```

When you refresh Community Builder, you see the vertical navigation menu with bullet points for each menu item. It uses the same dataset that drives the default navigation menu in a community. For this example, you don't want the vertical navigation menu to handle topic navigation. To remove the item, click the **Navigation Menu** button in the **Settings > Theme** area, and remove Topics from the navigation.

Now go back to the `verticalNav` menu and make it pretty. Here's the code for the completed component.

```
<aura:component extends="forceCommunity:navigationMenuBase">
  <div class="slds-grid slds-grid--vertical slds-navigation-list--vertical">
    <ul onclick="{!c.onClick}">
```

```

        <aura:iteration items="{!v.menuItems}" var="item">
            <li class="{!item.active ? 'slds-is-active' : ''}">
                <a href="javascript:void(0);" data-menu-item-id="{!item.id}"
class="slds-navigation-list--vertical__action slds-text-link--reset">
                    {!item.label}
                </a>
            </li>
        </aura:iteration>
    </ul>
</div>
</aura:component>

```

The example takes advantage of aura expression syntax to do some nifty things. You can conditionally add the `slds-is-active` class to the list item depending on whether the item is active. You also set the `data-menu-item-id` to be the item's unique ID, which you can use later to navigate to the corresponding item. In this way, you need only one click listener for the entire list, instead of adding one for each list item.

Add the click handler to the component's controller method. Note the JavaScript syntax for accessing data attributes on HTML elements, which allows you to get that item's ID.

```

({
    onClick : function(component, event, helper) {
        var id = event.target.dataset.menuItemId;
        if (id) {
            component.getSuper().navigate(id);
        }
    }
})

```

Add the following CSS rules to the theme layout component to remove unwanted margins and set the main content width.

```

.THIS .slds-col .ui-widget {
    margin: 16px 0;
}

.THIS .slds-col.content {
    width: 1140px;
}

```

SEE ALSO:

Lightning Component Developer Guide: [forceCommunity:navigationMenuBase](#)

Step 5: Build a Custom Search Component

Create a custom search component called `customSearch`, which implements the `forceCommunity:searchInterface`. This example queries several objects and returns record IDs that match our search term. Then you redirect to a custom page that contains the record names and links to the full record details.

1. Implement the `forceCommunity:searchInterface` Interface

Use a `<lightning:buttonIcon>` component and include a click handler.

```
<aura:component implements="forceCommunity:searchInterface">
<div class="slds-form-element slds-lookup" data-select="single">
  <div class="slds-form-element__control">
    <div class="slds-input-has-icon slds-input-has-icon--right">
      <lightning:buttonIcon iconName="utility:search" variant="bare" onclick="{!
c.handleClick }" alternativeText="Search" class="slds-input__icon" />
      <input type="search" class="slds-lookup__search-input slds-input"
placeholder="Search" />
    </div>
  </div>
</div>
</aura:component>
```

Add an attribute called `searchText` to contain the search text. Use a `<ui:inputText>` component instead of a plain `<input>` to bind the values.

```
<aura:attribute name="searchText" type="String" default=""/>
...
<ui:inputText value="{!v.searchText}" class="slds-lookup__search-input slds-input"
placeholder="Search" />
```

2. Create an Apex Controller

You need to create an Apex class for the component—let's call it `CustomSearchController`. Implement the method `searchForIds`, which takes a `String searchText` and returns a list of strings representing found IDs. For now, just return the search string itself.

```
public class CustomSearchController {
  @AuraEnabled
  public static List<String> searchForIds(String searchText) {
    return new List<String>{searchText};
  }
}
```

Specify this class as the controller for your component by adding it as the value for the controller attribute. Here's an example of the completed search component.

```
<aura:component implements="forceCommunity:searchInterface"
controller="CustomSearchController">
  <aura:attribute name="searchText" type="String" default=""/>
  <div class="slds-form-element slds-lookup" data-select="single">
    <div class="slds-form-element__control">
      <div class="slds-input-has-icon slds-input-has-icon--right">
        <lightning:buttonIcon iconName="utility:search" variant="bare" onclick="{!
c.handleClick }" alternativeText="Search" class="slds-input__icon" />
        <ui:inputText value="{!v.searchText}" class="slds-lookup__search-input slds-input"
placeholder="Search" />
      </div>
    </div>
  </div>
```

```

    </div>
</aura:component>

```

Now that you've hooked up the search component to an Apex controller, tell the component to execute that controller's action when the search button is clicked. Create a click handler for this component, and add a `handleClick` method. This example reads the value of the input text, sends it to the server-side Apex controller, and waits for a response. When you test the example, you see an array logged to the browser console.

```

({
  handleClick : function(component, event, helper) {
    var searchText = component.get('v.searchText');
    var action = component.get('c.searchForIds');
    action.setParams({searchText: searchText});
    action.setCallback(this, function(response) {
      var state = response.getState();
      if (state === 'SUCCESS') {
        var ids = response.getReturnValue();
        console.log(ids);
      }
    });

    $A.enqueueAction(action);
  }
})

```

3. Implement a Basic Search Query with SOQL

Now make the server controller do something more interesting. Salesforce supports the SOQL search language, which you can use in your Apex classes. For this query, take the input search text and try to find objects where that text appears in any field. Update the Apex class's method to return a list of record IDs for the accounts, campaigns, contacts, or leads that match the search term.

```

public static List<String> searchForIds(String searchText) {
    List<List<SObject>> results = [FIND :searchText IN ALL FIELDS RETURNING Account(Id),
    Campaign(Id), Contact(Id), Lead(Id)];
    List<String> ids = new List<String>();
    for (List<SObject> sobjs : results) {
        for (SObject subj : sobjs) {
            ids.add(subj.Id);
        }
    }
    return ids;
}

```

4. Return the Search Results to a Custom Page

Instead of just returning the record IDs, you can return the objects themselves or the IDs with extra information. You can even extend the search component to start searching after every key press and display partial results. For now, keep things simple and redirect to a new page that uses the IDs to display the record names and links to the full record details. You need two new components and a new custom page.

Create a component to show a single record. Based on the example for the Lightning data service, you can use this code.

```
<aura:component implements="force:hasRecordId" access="global">
  <aura:attribute name="record" type="Object"/>
  <aura:attribute name="simpleRecord" type="Object"/>
  <aura:attribute name="recordError" type="String"/>
  <force:recordData aura:id="recordLoader"
    recordId="{!v.recordId}"
    layoutType="COMPACT"
    targetRecord="{!v.record}"
    targetFields="{!v.simpleRecord}"
    targetError="{!v.recordError}"
    recordUpdated="{!c.handleRecordUpdated}" />

  <!-- Display a header with details about the record -->
  <div class="slds-page-header" role="banner">
    <p class="slds-text-heading--label">{!v.simpleRecord.Name}</p>
    <h1 class="slds-page-header__title slds-m-right--small slds-truncate
slds-align-left"><a href="{! $Site.siteUrlPrefix + '/detail/' + v.simpleRecord.Id}">Go to
details</a></h1>
  </div>

  <!-- Display Lightning Data Service errors, if any -->
  <aura:if isTrue="{!not(empty(v.recordError))}">
    <div class="recordError">
      <ui:message title="Error" severity="error" closable="true">
        {!v.recordError}
      </ui:message>
    </div>
  </aura:if>
</aura:component>
```

Create a drag-and-drop component called `customSearchResults`.

```
<aura:component implements="forceCommunity:availableForAllPageTypes" access="global">
  <aura:attribute type="list" name="recordIds" />
  <aura:handler name="init" value="{!this}" action="{!c.init}"/>
  <h1>Search Results</h1>
  <aura:iteration items="{!v.recordIds}" var="id">
    <c:customSearchResultItem recordId="{!id}"/>
  </aura:iteration>
</aura:component>
```

Create a controller. Here, you're relying on the record ID list to be passed to the component from the browser's session storage. This method allows data to be passed from page to page without affecting any URLs.

```
((
  init: function(component, event, helper) {
    var idsJson = sessionStorage.getItem('customSearch--recordIds');
    if (!$A.util.isUndefinedOrNull(idsJson)) {
      var ids = JSON.parse(idsJson);
      component.set('v.recordIds', ids);
      sessionStorage.removeItem('customSearch--recordIds');
    }
  }
})
```

In Community Builder, create a standard page called Custom Search Results, which produces a page URL of `custom-search-results`. Drag the **customSearchResults** component onto the page, along with whichever other customizations you want. You can even use the same custom theme layout that you created earlier in [Step 1: Create the Basic Theme Layout Structure](#), which the Home page is using.

Update the console log line in the original `customSearchController` JavaScript with code that sets the session storage value and fires a navigation event to the new page.

```
sessionStorage.setItem('customSearch--recordIds', JSON.stringify(ids));
var navEvt = $A.get('e.force:navigateToURL');
navEvt.setParams({url: '/custom-search-results'});
navEvt.fire();
```

In the CSS for the component, add the following CSS rules.

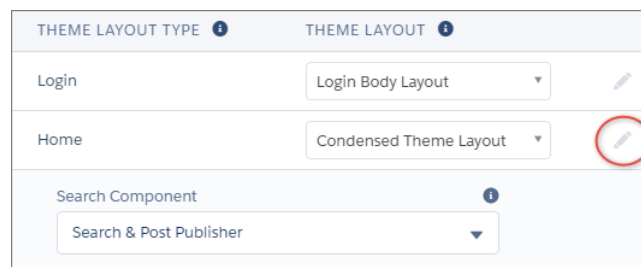
```
.THIS .slds-input__icon{
  margin-top: -.8rem;
}

.THIS {
  padding: 0 10px;
}
```

Add a label for the component in the bundle's design resource.

```
<design:component label="Custom Search">
</design:component>
```

In Community Builder, return to **Settings > Theme** and click the edit icon () to switch to the new Custom Search component.



If all went well, you can test out your new search component by entering a text string, clicking **Search**, and seeing which results show up!

SEE ALSO:

[Configure Swappable Search and Profile Menu Components](#)

[Salesforce Help: Create Custom Pages with Community Builder](#)

Step 6: Add Configuration Properties to the Theme Layout

Add an option that lets admins to hide the new search component completely in Community Builder.

In the theme layout component, add the following attribute.

```
<aura:attribute name="showSearch" type="Boolean" default="true" />
```



In the markup, wrap the entire search column with an `aura:if` expression. This expression is reactive, so when the attribute gets updated, the component rerenders.

```
<aura:if isTrue="{!v.showSearch}">
  <div class="slds-col">
    {!v.search}
  </div>
</aura:if>
```

Add the design attribute.

```
<design:component label="Condensed Theme Layout">
  <design:attribute name="showSearch" label="Show Search Box" />
</design:component>
```

In Community Builder, when you edit the Condensed Theme Layout, you now have an option to show or hide the search component. Deselecting the checkbox causes the page to rerender and hide the search component.

THEME LAYOUT TYPE ⓘ	THEME LAYOUT ⓘ
Login	Login Body Layout ▾ 
Home	Condensed Theme Layout ▾ 
<input checked="" type="checkbox"/> Show Search Box	
Search Component ⓘ Custom Search ▾	

CHAPTER 5 Develop Secure Code: LockerService and Stricter CSP

In this chapter ...

- [LockerService in Communities](#)
- [Critical Update for Stricter CSP Restrictions in Communities](#)

When you develop custom Lightning components or add head markup to your community, you need to be aware of LockerService and the stricter Content Security Policy (CSP) critical update. The LockerService architectural layer enhances security by isolating individual Lightning components in their own containers and enforcing coding best practices. The framework uses CSP to control the source of content that can be loaded on a page.

LockerService and CSP are documented in "[Developing Secure Code](#)" in the *Lightning Component Developer Guide*. Use that guide as your main point of reference for developing secure code.

LockerService is enforced the same way across all orgs. However, stricter CSP uses a separate critical update for Communities, which is documented more thoroughly [here](#).

LockerService in Communities

LockerService is a powerful security architecture for Lightning components that enhances security by isolating Lightning components in their own namespace. LockerService promotes best practices to improve the supportability of your code by allowing access only to supported APIs and eliminating access to non-published framework internals.

LockerService is enabled for all Lightning components set to API version 40.0 and later. API version 40.0 corresponds to Summer '17, when LockerService was enabled for *all* orgs. LockerService isn't enabled for components with API version 39.0 and earlier.

You can disable LockerService for a component by setting the component's API version 39.0 or earlier. However, for consistency and ease of debugging, we recommend that you set the same API version for all components in your app, when possible.

For information about working with LockerService when developing Lightning components, see [“What is LockerService?”](#) in the *Lightning Component Developer Guide*.

For information on preparing your Lightning components code for LockerService enablement, see [“Salesforce Lightning CLI \(Deprecated\).”](#)

SEE ALSO:


[Lightning Component Developer Guide: Browser Support Considerations for Lightning Components](#)

Critical Update for Stricter CSP Restrictions in Communities

The Lightning Component framework uses Content Security Policy (CSP), which is a W3C standard, to control the source of content that can be loaded on a page. The “Enable Stricter Content Security Policy for Lightning Components in Communities” critical update tightens CSP to mitigate the risk of cross-site scripting attacks.

Stricter CSP tightens CSP to mitigate the risk of cross-site scripting attacks by disallowing the `unsafe-inline` and `unsafe-eval` keywords for inline scripts (`script-src`). Ensure that your code and the third-party libraries that you use adhere to these rules by removing all calls using `eval()` or inline JavaScript code execution. You might have to update your third-party libraries to modern versions that don't depend on `unsafe-inline` or `unsafe-eval`.

In addition to affecting custom Lightning components, stricter CSP also affects the markup used in the `<head>` of your community's pages, when enabled. Inline scripts aren't permitted, and a warning appears when you enter unsupported markup tags in **Settings > Advanced** in Community Builder.

 **Note:** Stricter CSP was originally part of the LockerService critical update, which was automatically activated for all orgs in Summer '17. Stricter CSP was decoupled from LockerService in Summer '17 to give you more time to update your code.

Critical Update Timeline

The stricter CSP changes are available in two critical updates that affect only sandbox and Developer Edition orgs. The two critical updates—one for Communities and one for other contexts—are called:

- Enable Stricter Content Security Policy for Lightning Components in Communities
- Enable Stricter Content Security Policy for Lightning Components

Stricter CSP will gradually be available in more orgs. To understand the nuances between the two different critical updates, let's look at them together. Here's the planned timeline, but the schedule might change for future releases.

	Critical Update	Summer '17	Winter '18	Spring '18 (Feb 2018)	Summer '18	Winter '19 (Oct 2018)
Sandbox and DE orgs	Enable Stricter CSP for Lightning Components	OFF by default unless LockerService was activated in Spring '17				Activated for all orgs
	Enable Stricter CSP for Lightning Components in Communities	OFF by default				
Production orgs	Enable Stricter CSP for Lightning Components	N/A		ON by default		
	Enable Stricter CSP for Lightning Components in Communities	N/A		OFF by default		

Summer '17

The critical updates are available only in sandbox and Developer Edition orgs. Stricter CSP is not enforced in production orgs for this release.

Spring '18 (future plans)

The critical updates will be extended to all orgs, including production orgs.

- “Enable Stricter Content Security Policy for Lightning Components” will be enabled by default.
- “Enable Stricter Content Security Policy for Lightning Components in Communities” will be disabled by default.

You can activate and deactivate both critical updates as often as needed for testing purposes.

Winter '19 (future plans)

Both critical updates will be automatically activated for all orgs when the critical updates expire.

Activate “Enable Stricter Content Security Policy for Lightning Components in Communities”

In Communities, stricter CSP is disabled by default for sandboxes and Developer Edition orgs.

1. From Setup, enter *Critical Updates* in the Quick Find box, and then select **Critical Updates**.
2. For “Enable Stricter Content Security Policy for Lightning Components in Communities”, click **Activate**.
3. Refresh your browser page.

What Does This Critical Update Affect?

This critical update enables stricter CSP in sandboxes and Developer Edition orgs for Communities only.

The critical update doesn't affect:

- Salesforce Classic
- Any apps for Salesforce Classic, such as Salesforce Console in Salesforce Classic
- Lightning Out, which allows you to run Lightning components in a container outside of Lightning apps, such as Lightning components in Visualforce and Visualforce-based Communities. The container defines the CSP rules.

 **Tip:** To enable stricter CSP for Lightning Experience, the Salesforce app, and standalone apps, use the “Enable Stricter Content Security Policy for Lightning Components” critical update.


SEE ALSO:

[Lightning Component Developer Guide: Content Security Policy Overview](#)

[Lightning Component Developer Guide: Critical Update for Stricter CSP Restrictions](#)

CHAPTER 6 Analyze and Improve Community Performance

The Salesforce Community Page Optimizer analyzes your community and identifies issues that impact performance. Use the information to refine your design and improve community performance for your members. The Page Optimizer is a free plug-in available from the Chrome Web Store. Download and install the plug-in as you would any Chrome extension.

To download the Community Page Optimizer, in Community Builder, click  on the left sidebar and click **Advanced**.

EDITIONS

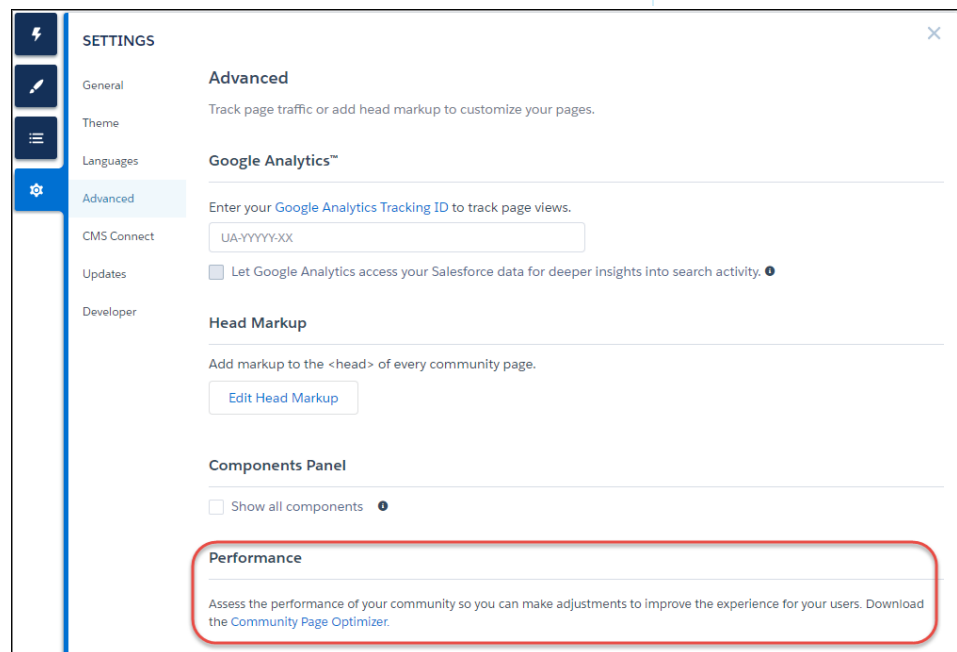
Available in: Salesforce Classic and Lightning Experience

Available in: **Enterprise, Performance, Unlimited,** and **Developer** Editions

USER PERMISSIONS

To create, customize, or publish a community:

- Create and Set Up Communities AND View Setup and Configuration

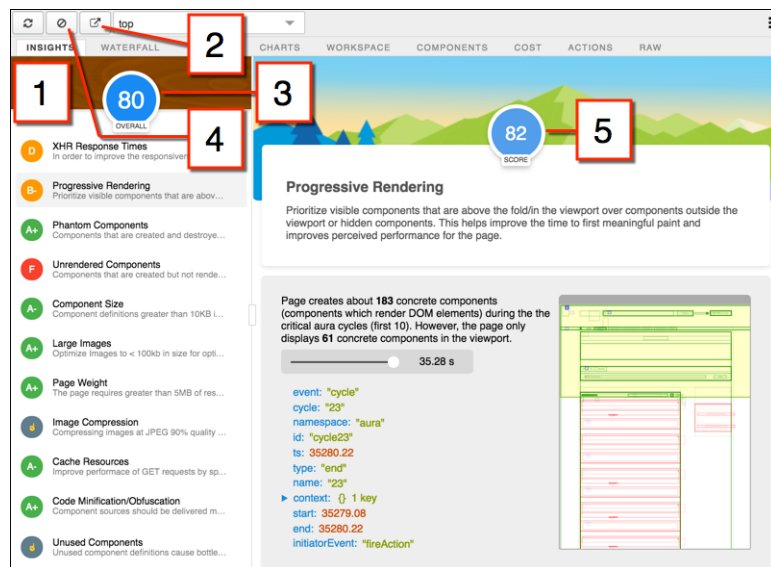


After installation, the Community Page Optimizer is located with your other Chrome extensions.



Insights

To analyze your community, navigate to your published community, load the page, and then launch the Community Page Optimizer.



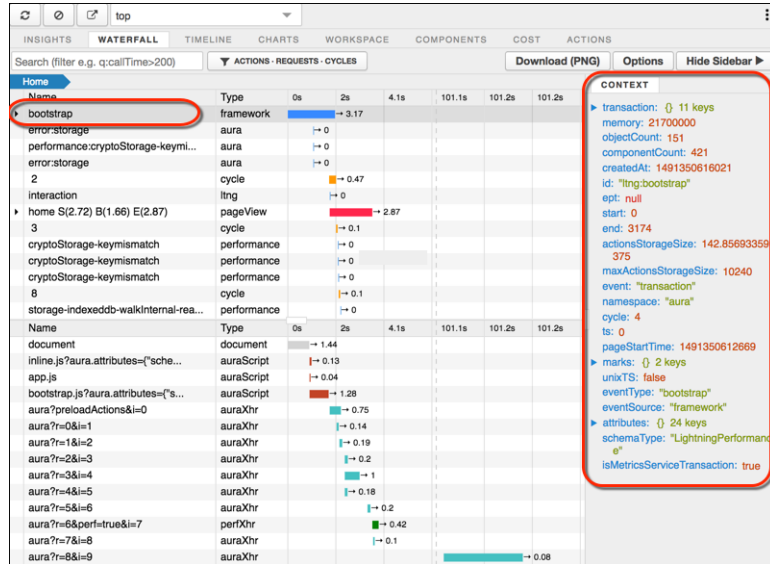
The Insights tab (1) evaluates your page based on best practices for web applications developed using the Lightning framework. This tab displays an overall performance score (3) along with individual scores (5) for various analysis rules. To view details and suggested actions, click each rule. Click **Popout** (2) for more room to work.

The Insights tab is conservative in providing recommendations. For further insights, consider reviewing the raw data presented on the Waterfall, Timeline, Charts, Cost, and Actions tabs.

Click **Clear** (4) to remove collected metrics. Perform some user actions on the page to collect new metrics and then reopen the Community Page Optimizer. For example, to gather performance metrics for liking a feed item, clear performance metrics, click Like, and reopen the Community Page Optimizer.

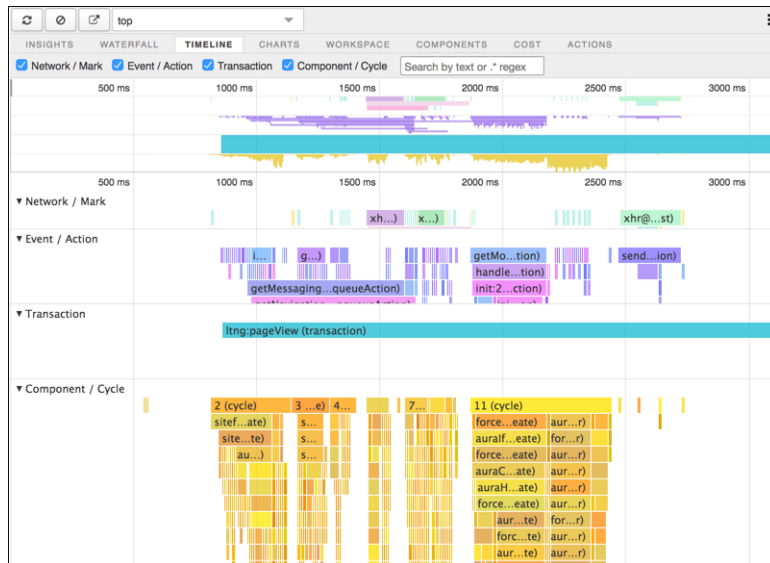
Waterfall

The Waterfall tab displays all network requests and performance instrumentation data. Click a row to view contextual information in the sidebar. Click the arrow to the left of each row to expand the information for each row.



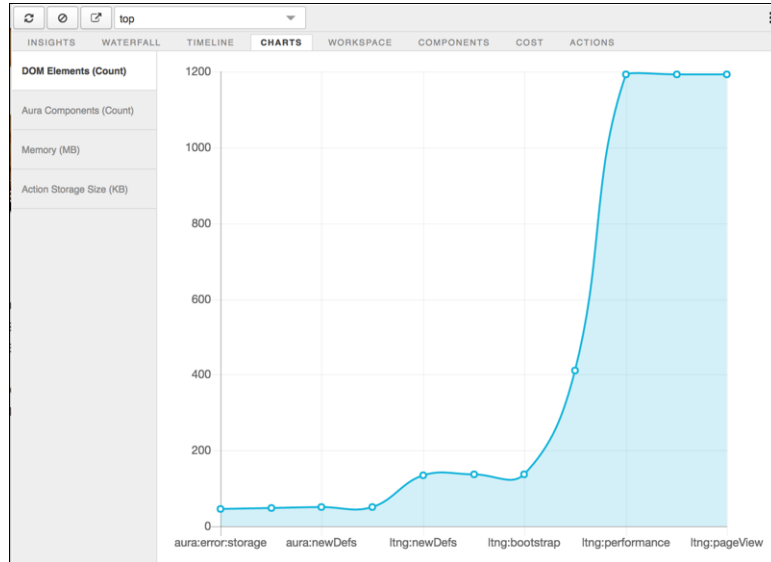
Timeline

The Timeline tab provides a profile of each component's rendering life cycle. The timeline view is optimized for displaying Lightning framework metrics, so it's easier to interpret than Chrome DevTools.



Charts

The Charts tab displays trending information about memory and components as customers use your page.



Components

The Components tab displays the life cycle counts for each component on the page. This view helps you identify potential component leaks and unexpected rendering behavior. Use the Component tab along with the Cost tab for an overall view of component performance.

Id	Name	Create	Render	Rerender	Unrender	AfterRender	Destroy
-	siteforce:napiApp	1	1	-	-	1	-
-	siteforce:baseApp	1	1	-	-	1	-
-	siteforce:routerInitializer	1	1	-	-	1	-
-	force:toastManager	1	1	-	-	1	-
-	force:toastMessageQueue	1	1	-	-	1	-
-	force:hoverPrototypeManager	1	1	-	-	1	-
-	one:actionsManager	1	1	-	-	1	-
-	force:targetInteractionHandler	1	1	-	-	1	-
-	siteforce:panelsContainer	1	1	-	-	1	-
-	siteforce:spinnerManager	1	1	-	-	1	-
-	siteforce:loadingBalls	2	2	-	-	2	-
-	siteforce:panelManager	1	1	-	-	1	-
-	one:panelManager	1	1	-	-	1	-
-	forceContent:filesManager	1	1	-	-	1	-
-	forceContent:modalPreviewManager	1	1	-	-	1	-
-	force:hostConfig	1	1	2	-	1	-
-	siteforce:qb	1	1	-	-	1	-
-	instrumentation:beacon	1	1	-	-	1	-
-	force:quickActionManager	1	1	-	-	1	-
-	notes:editPanelManager	1	1	-	-	1	-

Cost

The Cost tab displays the amount of time each component was busy processing its logic. The lower the time, the better the performance.

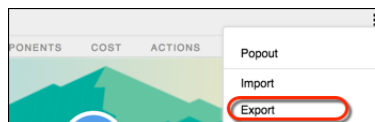
		Self			Aggregate	
Name	Count	Average	Total		Average	Total
siteforceNapiliApp	1	24.23ms	24.23ms 1.83%		303.6ms	303.6ms
siteforceBaseApp	1	0.92ms	0.92ms 0.07%		279.37ms	279.37ms
siteforceRouterInitializer	1	12.05ms	12.05ms 0.91%		12.71ms	12.71ms
auraComponent	372	0.48ms	84.59ms 6.37%	43.4ms (nested)	3,905.22ms (nested)	
uiAsyncComponentManager	1	1.37ms	1.37ms 0.10%		2.06ms	2.06ms
uiContainerManager	1	2.85ms	2.85ms 0.21%		9.37ms	9.37ms
auraHtml	684	0.8ms	232.07ms 17.48%	40.75ms (nested)	6,188.22ms (nested)	
forceToastManager	1	1.39ms	1.39ms 0.10%		12.11ms	12.11ms
forceToastMessageQueue	1	4.84ms	4.84ms 0.36%		9.49ms	9.49ms
auralteration	19	1.98ms	33.26ms 2.51%	41.88ms (nested)	655.32ms (nested)	
auraExpression	327	0.56ms	65.44ms 4.93%	15.46ms (nested)	4,813.62ms (nested)	
auralf	340	0.73ms	175.6ms 13.23%	21.04ms (nested)	2,387.85ms (nested)	
forceHoverPrototypeMana...	1	5.34ms	5.34ms 0.40%		8.03ms	8.03ms
forceHoverPrototype	1	1.95ms	1.95ms 0.15%		2.35ms	2.35ms
oneActionsManager	1	2.11ms	2.11ms 0.16%		5.7ms	5.7ms
forceTargetInteractionHand...	1	3.26ms	3.26ms 0.25%		3.44ms	3.44ms
siteforcePanelsContainer	1	0.53ms	0.53ms 0.04%		10.07ms	10.07ms
siteforceSpinnerManager	1	0.68ms	0.68ms 0.05%		3.5ms	3.5ms
siteforceLoadingBalls	2	1.67ms	3.33ms 0.25%		3.47ms	6.94ms
siteforcePanelManager	1	1.48ms	1.48ms 0.11%		5.75ms	5.75ms
onePanelManager	1	2.66ms	2.66ms 0.20%		4.14ms	4.14ms
uiPanelManager2	1	0.71ms	0.71ms 0.05%		1.48ms	1.48ms
forceContentFilesManager	1	3.59ms	3.59ms 0.27%		8.01ms	8.01ms

Actions

The Actions tab displays a list of all actions performed on the page, along with their timing information.

Export

Export your analysis to a file to share with your development and support teams.



Submit Feedback

We want to hear from you. Share your comments, questions, requests, and any issues that you find. [Submit Feedback](#).

SEE ALSO:

[Salesforce Developer Blog: Lightning Components Performance Best Practices](#)

CHAPTER 7 Connect Your Community to Your Content Management System

In this chapter ...

- [Before Using CMS Connect](#)
- [Create a CMS Connection](#)
- [Edit a CMS Connection](#)
- [Manage CMS Connections](#)
- [Add CMS Content to Your Community Pages](#)
- [Personalize Your CMS Content](#)
- [CMS Connect Recommendations for Optimal Usage](#)
- [CMS Connect Examples](#)

CMS Connect is a tool for embedding content from a third-party content management system (CMS) in your Salesforce community. You can connect HTML, JSON (Beta), CSS, and JavaScript to customize your community and keep its branding and other content consistent with your website.



Note: This release contains beta versions of some features in CMS Connect, which means it's a high-quality feature with known limitations. CMS Connect isn't generally available unless or until Salesforce announces its general availability in documentation or in press releases or public statements. 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. You can provide feedback and suggestions for CMS Connect in the Community Cloud group in the Trailblazer Community.

After you do some initial configuration work, CMS Connect makes maintenance a breeze, because your content renders dynamically on your community pages. If you have website content in AEM, Drupal, SDL, Sitecore, or WordPress, CMS Connect is the smart way to display headers, footers, banners, blogs, articles, and other reused content in your community. We give you many configuration options including language mapping between your CMS and Salesforce, determining the load order of multiple connections, and specifying CSS scope.

CMS Connect is available in communities that are based on Customer Service (Napili), Partner Central, and Lightning Bolt solutions.

CMS Connect supports the following content management systems:

- Adobe Experience Manager (AEM)
- Drupal
- SDL
- Sitecore
- WordPress

Before Using CMS Connect

Ready to get your CMS and your community connected? Before diving in, review these pointers and prerequisites so everything goes smoothly.

Your HTTP server must serve HTML fragments

CMS Connect requires an HTTP server that can serve HTML fragments, either static or rendered on demand. Fragments can include headers, footers, components, CSS, or JavaScript.

URLs in CSS and JavaScript must be absolute

URLs in CSS and JavaScript must be absolute. Relative URLs in HTML are okay and are converted for you. CMS Connect appends host names and converts relative URLs to absolute URLs in the following HTML tags and attributes:

- `` tag `src` attribute
- `<audio>` tag `src` attribute
- `<input>` tag
- `<button>` tag `formaction` attribute
- `<video>` tag `src` and `poster` attributes
- `<a>` and `<area>` tags `href` attribute
- `<form>` tag `action` attribute
- ``, `<ins>`, `<blockquote>`, and `<q>` tags `cite` attribute
- `<script>` tag `src` attribute

Community Host must be a trusted host in the Cross-Origin Resource Sharing (CORS) header

CMS Connect uses Cross-Origin Resource Sharing (CORS) to access external content. Make sure to add Community Host to the list of trusted hosts in the CORS header in your CMS system.

CORS is a web standard for accessing web resources on different domains. CORS is a required technology to connect your CMS to Salesforce. It's a technique for relaxing the same-origin policy, allowing JavaScript on a web page to consume a REST API served from a different origin. CORS allows JavaScript to pass data to the servers at Salesforce using CMS Connect.

To enable CORS in development environments, we recommend using a [Chrome plugin](#). For production environments, please visit your CMS documentation on enabling CORS.

For more information about CORS, see https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS.

Some tags and scripts aren't allowed

CMS Connect filters out the same HTML tags that Locker Service and Lightning Components do. Get familiar with these now to avoid surprises later. See "Add Markup to the Page `<head>` to Customize Your Community" in the Salesforce online help for a list of supported tags.

All CMS servers must be accessible via unauthenticated HTTPS (HTTP over SSL)

All CMS servers you connect must be accessible via unauthenticated HTTPS (HTTP over SSL) to retrieve HTML and JavaScript. When you set up a CMS connection, the server URL you enter must start with HTTPS. This is to ensure all web communications that are required remain private. An SSL certificate is required for unauthenticated HTTPS for all traffic between your servers and Salesforce.

All JavaScript and CSS must be from the same source as HTML

All JavaScript and CSS files referenced by your HTML must point to your CMS source.

Community Workspaces must be enabled

To use CMS Connect, you must have Community Workspaces enabled in your Community Settings. From Setup, go to Community Settings. Make sure the Enable Community Workspaces checkbox is selected.

CMS Connect org perm must be left on


CMS Connect is controlled by an org permission that is turned on by default. If you're not seeing CMS Connect in your Community Workspaces, it's possible that the permission is turned off. You can ask Salesforce Customer Support to turn it back on for you.

Create a CMS Connection

Create a connection between your content management system and your community so you can render headers, footers, banners, blogs, and other content on your community pages.


Read [Before Using CMS Connect](#) to make sure you're ready to connect to your CMS.

1. Go to Community Workspaces.
2. Click **CMS Connect**.
3. Click **New CMS Connection** (if no connections have been created yet in your community) or **New**.
4. For **Name**, enter a friendly name for the connection. The name shows up in your CMS workspace and other internal areas. (An API name is created for the connection behind the scenes, based on the name you enter.)
5. Select your CMS source: AEM, Drupal, SDL, Sitecore, Wordpress, or Other.


 **Note:** The "Other" option isn't fully supported. However, if your CMS server isn't listed, CMS Connect works if you set it up properly. CMS Connect works with the HTML, CSS, and HTTP standards and isn't provider-specific.

6. For your server URL, enter the full path to your CMS server, such as: `https://www.example.com`.
7. For Root Path, enter the path to the directory that your CMS content is in. You can include placeholders for language and component. For example, a root path to content in AEM might look like this:

```
content/mywebsite/{language}/{component}
```

 **Note:** The `{language}` placeholder isn't required, but if you include it in your root path, enable language mapping and add at least one language. See [Build a CMS Connect Root Path and Component Paths](#) for details on root paths and how they work with component paths.

- If your CMS source is AEM and your HTML content is set up with personalization, you can use that personalized content in your community. To do that, enable **Use Personalization**. See [Personalize Your CMS Content](#) for details on setting up personalization.

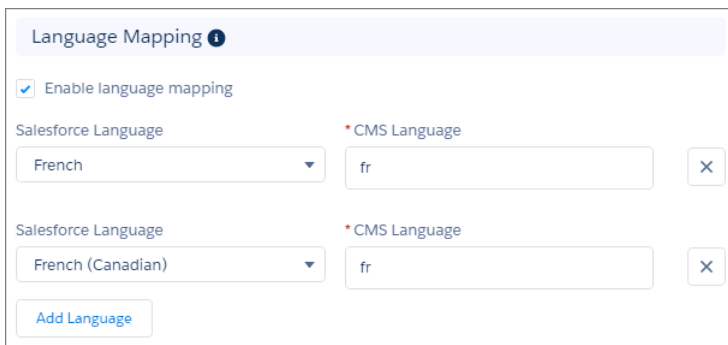
 **Note:** To use personalization, enable it for the components you want to personalize. Do that in Builder Settings for header and footer, and in Community Builder for banners or other components.

- To include CSS, click **Add CSS** to add one or more URLs to your CSS files. If your CSS is scoped, you can specify it in the Scope field. Stylesheets load in the order listed. Use the up and down arrows to change the order.
- To include JavaScript, click **Add Script** to add one or more URLs to your JavaScript files. Scripts load in the order listed. Use the up and down arrows to change the order.
- To connect JSON content such as blogs, click **Add JSON** and enter a name, type, and path for each JSON component you want to add. See [Set Up JSON in Your CMS Connection \(Beta\)](#) for details.
- If your content has multiple languages, select **Enable language mapping**. See [Set Up Language Mapping in Your CMS Connection](#) for more information on setting up language mapping.

In the Salesforce Language dropdown, English is selected by default. To map English, for CMS Language, enter the directory name of your English language folder from AEM. For example, enter **en**.

To add more languages, click **Add Language**. For each language that you add from your CMS, make sure it's enabled in your Builder Settings.

If you want to map languages from your community that you don't have in your CMS, you can define what language the CMS content should display in. For example, if your community has French and French Canadian enabled, you can set it up like this, so the French Canadian community displays French content:



- Click **Save**.

[Build a CMS Connect Root Path and Component Paths](#)

Entering a root path when configuring a CMS connection saves time when adding CMS Connect components to your pages in Community Builder. A root path uses placeholders for the common parts of content URLs. Root paths from Adobe Experience Manager, for example, start with `/content`. Instead of entering the full path for each component, you only need to enter the component-specific part of the path.

[Set Up Language Mapping in Your CMS Connection](#)


Language Mapping allows you to have copies of your entire site in other languages. It doesn't matter how your languages are named in your CMS. Use language mapping to configure a mapping to Salesforce languages.

[Set Up JSON in Your CMS Connection \(Beta\)](#)

Does your website have JSON content such as blogs or articles? Do you store it in a CMS such as WordPress or Drupal? Sweet! You can render this content in your community using CMS Connect.

Build a CMS Connect Root Path and Component Paths

Entering a root path when configuring a CMS connection saves time when adding CMS Connect components to your pages in Community Builder. A root path uses placeholders for the common parts of content URLs. Root paths from Adobe Experience Manager, for example, start with `/content`. Instead of entering the full path for each component, you only need to enter the component-specific part of the path.

 **Example:** The full path to a banner component is:

```
content/capricorn/{language}/banner.html
```

Enter `content/capricorn/{language}` for the root path, and `/banner.html` for the component path in the Builder.

You don't have to enter a root path when setting up a connection. If you leave it blank, you'll just need to enter full paths for your header, footer, and components (content fragments).

You can include placeholders for language and component in your root path. If you don't include them, we add them in invisibly.

 **Example:** This root path:

```
content/capricorn
```

is read as

```
content/capricorn/{language}/{component}
```

If you include the `{language}` placeholder explicitly, you'll be required to enable language mapping and enter at least one language in order to save your connection. If you don't include the placeholder, you can still enable language mapping and add languages.

Root paths can point only to HTML. They can't point to JPG files.

Set Up Language Mapping in Your CMS Connection

Language Mapping allows you to have copies of your entire site in other languages. It doesn't matter how your languages are named in your CMS. Use language mapping to configure a mapping to Salesforce languages.

In AEM, language names are in the directory `/content/projectname/{language}`.

 **Example:** The directory for French is:

```
/content/projectname/{language}/fr
```

So `fr` is the CMS Language, which you can map to the Salesforce language `French`.

Make sure all mapped languages are enabled in your community

Let's say you've mapped all the languages from your CMS to Salesforce languages. Great! But that doesn't necessarily mean all those languages are enabled in your community. Check your Builder Settings to be sure. If any are missing, just add them. Do this on Site.com.

Map languages enabled in your community but not in your CMS

Your community might have some languages enabled that don't exist in your CMS. You can map these too, to define which language the CMS content displays in.

Set Up JSON in Your CMS Connection (Beta)

Does your website have JSON content such as blogs or articles? Do you store it in a CMS such as WordPress or Drupal? Sweet! You can render this content in your community using CMS Connect.

CMS Connect supports two basic types of JSON content. What they're called in your CMS might be different than what another CMS calls them. In CMS Connect, to keep things simple, we call them *Content Item* and *Content List*.

An example of a content item is a single blog post. When it displays on a page, it's the full blog post, not just a blurb about it. A content list, on the other hand, is a grouping of items such as blog posts. Most often, each item in a content list contains a link to those items. When setting up paths to your JSON content in a CMS connection, specify the type for each one: content item or content list.

1. When [creating](#) or [editing](#) a CMS connection, in the JSON section, click **Add JSON**.
2. Enter a name for your content. It can be anything you want. For example, *Home Improvement*.
3. Select the type of content: **Content List** or **Content Item**. Is it a single article or blog post, such as "DIY Dryer Vent Cleaning"? Then it's a content item. Or is it a grouping of items, such as "Home Improvement", that has links to individual blog posts? Then it's a content list.
4. Enter the path to the JSON component in your CMS.
5. Want to add more JSON? Repeat the above steps for each content item or content list to add.
6. Click **Save** when finished to save the connection.

For instructions on adding JSON content to your community pages, see [Add CMS Connect \(JSON\) Components to Your Community Pages \(Beta\)](#)



Example: Here's an example of a JSON content item in a CMS connection that uses Drupal. The path to the content item is structured as follows: `{baseUrl}/jsonApi/node/{contentType}/{id}`

Here's what the JSON payload structure for the content item looks like:

```
{
  "data": {
    "type": "node--page",
    "id": "c53cf56c-f70d-456e-838b-47788742b074",
    "attributes": {
      "nid": 5,
      "uuid": "c53cf56c-f70d-456e-838b-47788742 b074",
      "vid ": 5,
      "langcode": "en",
      "title": "This is an Example.",
      "created": 1502133909,
      "changed": 1502133933,
      "body": {
        "value": "This is the body.",
        "summary ": ""
      }
    },
    "relationships": {
      "type ": {
        "data ": {
          "type": "node_type--node_type",
          "id": "5b80bc9e-dc78-4612-add8-e46b2e2ff616"
        }
      }
    }
  }
}
```

```
}
}
```

This example is taken from a Drupal CMS where @data is the parent node, and all attribute nodes to @data can be included in JSON expressions.

 **Note:** All JSON data sources must have 1 parent node. Multiple parent nodes in the JSON structure causes an error. For more information on constructing a data source that meets these criteria, see your JSON API options in your CMS.

An example JSON expression for retrieving the title of the content is:

```
@data/attributes/title
```

An example JSON expression for retrieving the body of the content is:

```
@data/attributes/body/value
```

Your JSON expressions can be retrieved from multiple data sources and included in your community by creating JSON endpoints in your CMS Connect workspace. Your JSON expressions can handle any node depth.

 **Example:** Here's an example of a JSON content list in a CMS connection that uses Drupal. The path to the content list is structured as follows: {baseUrl}/jsonApi/node/page

Here's what the JSON payload structure for the content list looks like:

```
{
  "data": {
    "type": "node--page",
    "id": "c53cf56c-f70d-456e-838b-47788742b074",
    "attributes": {
      "nid": 5,
      "uuid": "c53cf56c-f70d-456e-838b-47788742b074",
      "vid": 5,
      "langcode": "en",
      "status": true,
      "title": "Test",
      "created": 1502133909,
      "changed": 1502133933,
      "promote": false,
      "sticky": false,
      "revision_timestamp": 1502133933,
      "revision_log": null,
      "revision_translation_affected": true,
      "default_langcode": true,
      "path": null,
      "body": {
        "value": "<p>Here is the header</p>\r\n",
        "format": "basic_html",
        "summary": ""
      }
    }
  },
  "relationships": {
    "type": {
      "data": {
        "type": "node_type--node_type",
        "id": "5b80bc9e-dc78-4612-add8-e46b2e2ff616"
      }
    }
  }
}
```



```
    },
    "links": {
      "self":
"https://www.sandbox7.net/jsonapi/node/page/c53cf56c-f70d-456e-838b-47788742b074/relationships/type",

      "related":
"https://www.sandbox7.net/jsonapi/node/page/c53cf56c-f70d-456e-838b-47788742b074/type"

    }
  },
  "uid": {
    "data": {
      "type": "user--user",
      "id": "d5808807-9f3d-4f10-a031-c3340172b88e"
    },
    "links": {
      "self":
"https://www.sandbox7.net/jsonapi/node/page/c53cf56c-f70d-456e-838b-47788742b074/relationships/uid",

      "related":
"https://www.sandbox7.net/jsonapi/node/page/c53cf56c-f70d-456e-838b-47788742b074/uid"

    }
  },
  "revision_uid": {
    "data": {
      "type": "user--user",
      "id": "d5808807-9f3d-4f10-a031-c3340172b88e"
    },
    "links": {
      "self":
"https://www.sandbox7.net/jsonapi/node/page/c53cf56c-f70d-456e-838b-47788742b074/relationships/revision_uid",

      "related":
"https://www.sandbox7.net/jsonapi/node/page/c53cf56c-f70d-456e-838b-47788742b074/revision_uid"

    }
  }
},
"links": {
  "self":
"https://www.sandbox7.net/jsonapi/node/page/c53cf56c-f70d-456e-838b-47788742b074"

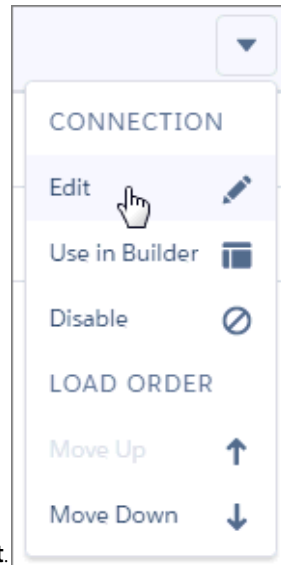
}
},
"links": {
  "self":
"https://www.sandbox7.net/jsonapi/node/page/c53cf56c-f70d-456e-838b-47788742b074"

}
}
}
```

Edit a CMS Connection

You can edit a CMS connection that's already been set up in your community. For example, change the language mapping, or add CSS and JavaScript files.

1. Open [Community Workspaces](#).
2. Click **CMS Connect**.
- 3.



Click for the connection you want to edit. Choose **Edit**.

4. Make changes as needed. See [Create a CMS Connection](#) for details.
5. Click **Save**.

Manage CMS Connections

In your CMS Connect workspace, you can enable and disable connections and change their load order.

Change the load order of CMS connections

If your community has multiple CMS connections, you can decide the order in which they're loaded. The order mostly affects any CSS and JavaScript in your connections. Consider their dependencies on each other, and set the load order accordingly.

For example, suppose one of your connections has the JavaScript library jquery, and another connection relies on jquery. Set the connection with jquery to load first so that the other one can load.

Header and footer always render first regardless of the load order of your connections.

Disable and enable CMS connections

You can't delete a CMS connection once it's been created, but you can disable it. Disabling a connection means:

- Its content isn't rendered
- Its load order is ignored when connections are loaded

If you try to add content to a page for a connection that's disabled, you'll get an error message.

1. Open [Community Workspaces](#).
2. Click **CMS Connect**.
3. Click for the connection you want, and choose **Enable** or **Disable**.

Add CMS Content to Your Community Pages

Come one, come all! Your headers, footers, banners, blogs, HTML, JSON, and other content from your CMS are welcome on your community pages.

[Add CMS Header and Footer Components to Your Community](#)

Once you've set up your CMS connection and added a header and footer, you're ready to add components to your pages.

[Add CMS Connect \(HTML\) Components to Your Community Pages](#)

Once you've set up your CMS connection and added a header and footer, you're ready to add components to your pages.

[Add CMS Connect \(JSON\) Components to Your Community Pages \(Beta\)](#)

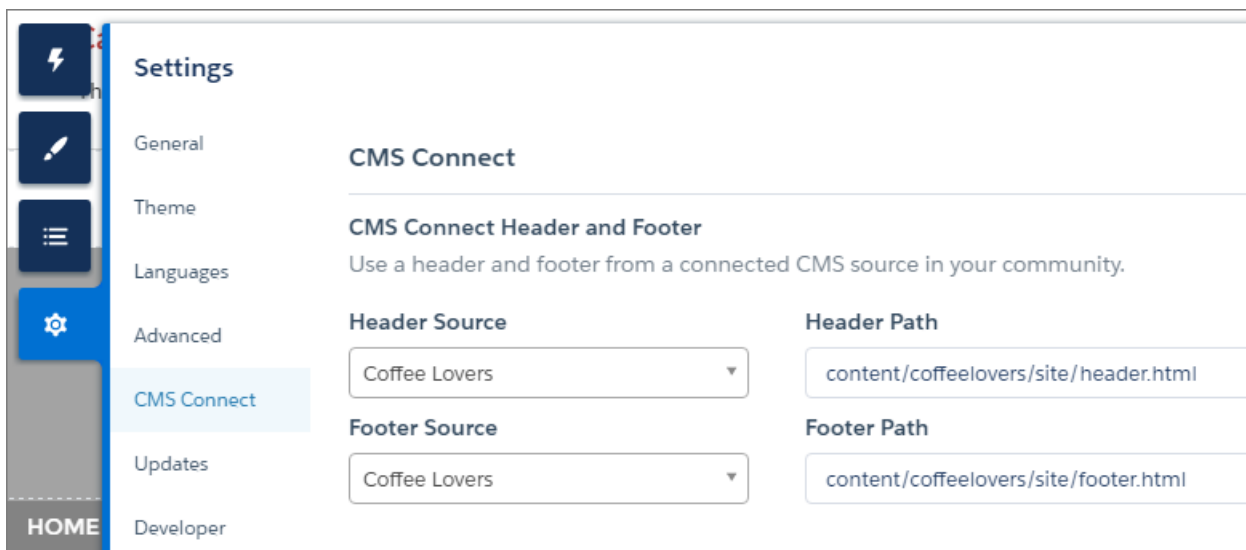
Use CMS Connect to render your JSON content, such as articles and blogs. Use the property editor in the Community Builder to customize layouts to display your content just the way you want.

Add CMS Header and Footer Components to Your Community

Once you've set up your CMS connection and added a header and footer, you're ready to add components to your pages.

Before you can add CMS headers and footers, [set up your CMS connection](#) on page 56.


1. From Community Builder, go to **Settings**.
2. Click **CMS Connect**.
3. Select a header source and enter a header path.
4. Select a footer source and enter a footer path.



Add CMS Connect (HTML) Components to Your Community Pages

Once you've set up your CMS connection and added a header and footer, you're ready to add components to your pages.


Before you can add HTML components to your community pages, [set up your CMS connection](#) on page 56.

1. Open [Community Workspaces](#).
2. Click **CMS Connect**.
3. Click  for the connection that has the components you want to add.
4. Choose **Use in Builder**.
5. Navigate to the page you want.
6. Drag a CMS Connect (HTML) component to the place on the page where you want to display it.
7. Select the component. In the property editor, configure its properties.

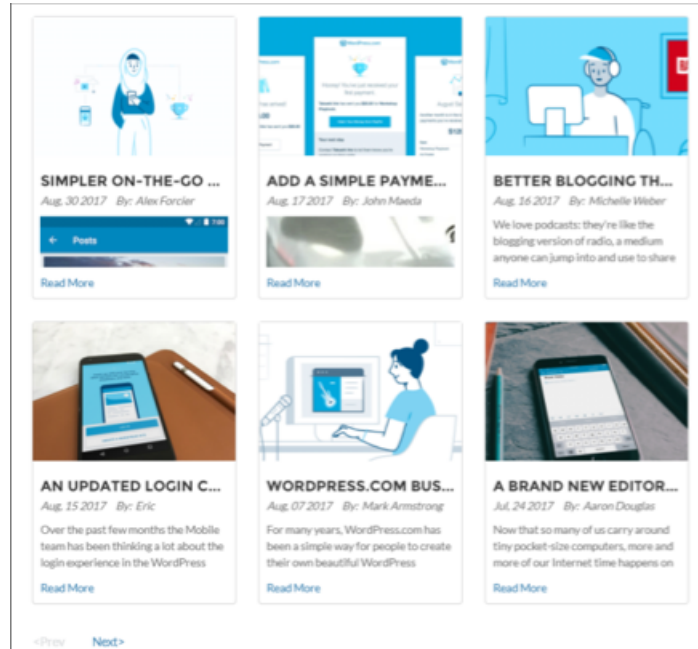
Add CMS Connect (JSON) Components to Your Community Pages (Beta)

Use CMS Connect to render your JSON content, such as articles and blogs. Use the property editor in the Community Builder to customize layouts to display your content just the way you want.

Before you can add JSON to your community pages, [set up JSON in your CMS connection](#) on page 59.

1. Open [Community Workspaces](#).
2. Click **CMS Connect**.
3. Click  for the connection that has the components you want to add.
4. Choose **Use in Builder**.
5. Navigate to the page you want.
6. Drag a CMS Connect (JSON) component to the place on the page where you want to display it.
7. Select the component. In the property editor, [configure its properties](#).

 **Example:** Let's say you want to display a list of blogs on your page.



The property editor will look something like:

CMS CONNECT (JSON) ✕

BETA

Display JSON content from your CMS in your community.

Connection ⓘ

CMS Source
Wordpress ▼

JSON Content
Blogs ▼

Component Path

Content List Node Path ⓘ
@posts

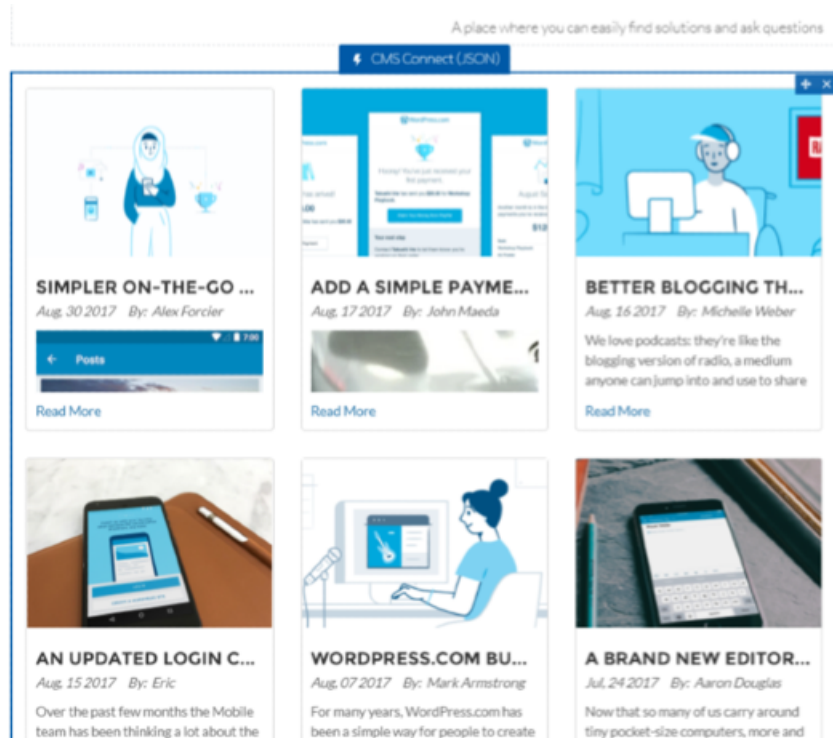
Content List Layout ⓘ

Content List Layout
Grid ▼

* Items Per Page ⓘ
6

* Columns ⓘ
3

After you save the JSON settings in the property editor, a preview of the content list displays in the page area:



Note: This release contains a beta version of the JSON connector, which means it's a high-quality feature with known limitations. The JSON connector isn't generally available unless or until Salesforce announces its general availability in documentation or in press releases or public statements. 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. You can provide feedback and suggestions for CMS Connect in the Community Cloud group in the Trailblazer Community.

Personalize Your CMS Content

CMS Connect supports content from Adobe Experience Manager (AEM) that is personalized using Client Context. If you have content in AEM that is personalized using Client Context, you can enable personalization in your community so you decide who sees what. Personalization in CMS Connect lets you keep the branding and other personalized content consistent between your community and your website. Render content according to different segments of users, based on criteria such as geolocation or language.

Some upfront effort is required to get personalization working in your community. You need to create and install a connector JSP page and expose it through an HTML page in AEM. The connector page contains the JSP with your website's personalization mapping logic. We provide the code for it in [CMS Connector Page Code](#). You might need to add some code, depending on how you want to run scripts. Then provide the path to this connector page in AEM when you're setting up the CMS connection in your community. In your CMS connection, you can also add a path to your JavaScript file if you want to run scripts dynamically inside the JSP file.

Ready to get started? Let's dig in. (Take a power nap first, if you need to.)

Set Up Personalization and the Connector Page in AEM

1. If you haven't done so already, set up personalization using Client Context in AEM.
 - Create personalization rules based on your segments and what you want them each to see. Create an experience for each segment.

- Determine which personalized content you want to host in your community. Each component in AEM has a default URL. Make a list of these URLs, along with the components they're for. You'll need these when you set up your CMS connection.
2. Use the [CMS connector page code](#) on page 70 to create a connector JSP page and expose it through an HTML page in AEM.

Enable Personalization in Your CMS Connection


1. [Create a CMS connection](#) on page 56 (or [edit an existing one](#) on page 62) where you want to host your personalized content.

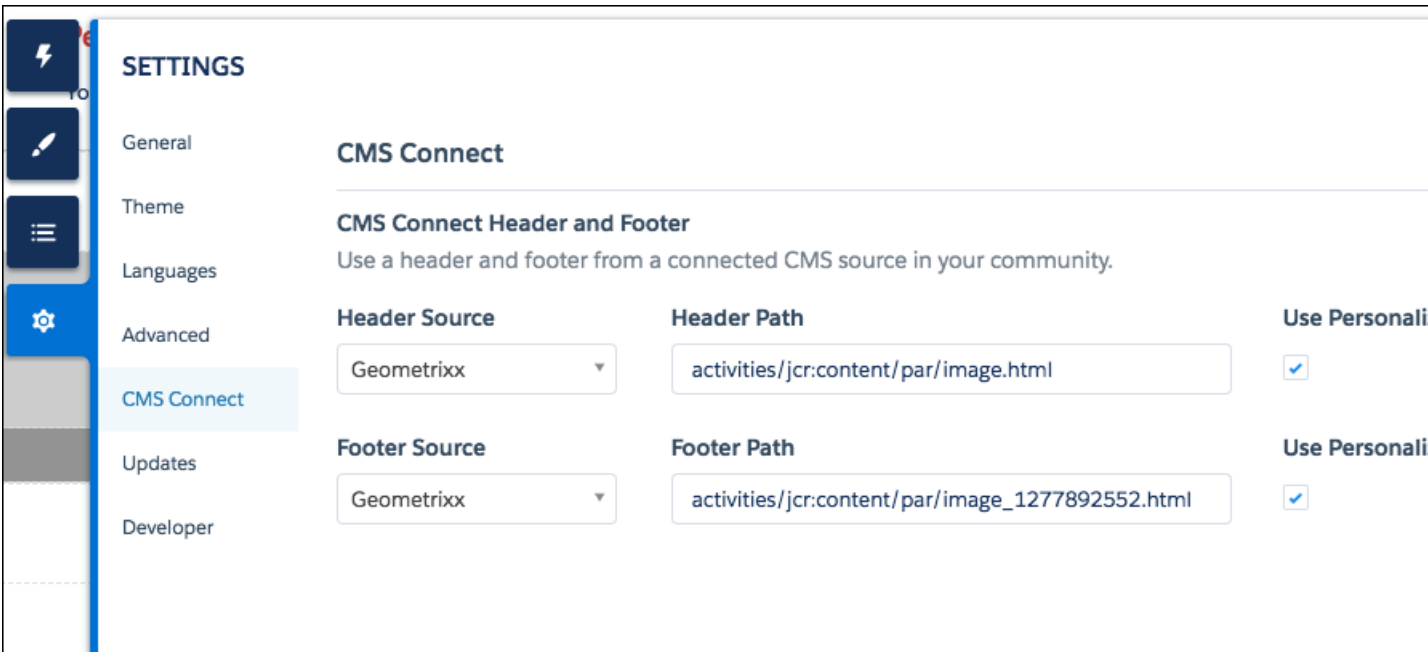
The screenshot shows the 'Edit Geometrixx' configuration form. The form is titled 'Edit Geometrixx' and contains several fields and a checkbox. The fields are: 'Name' (Geometrixx), 'CMS Source' (AEM), 'Server URL' (https://www.myaemserver.com), 'HTML', 'Root Path' (content/geometrixx-outdoors/{language}), 'Connector Page Path' (content/sfdcConnector.html), 'Script Path' (etc/segmentation/geometrixx-outdoors.segment.js), and 'CSS'. There is a checkbox for 'Use Personalization' which is checked. At the bottom right, there are 'Cancel' and 'Save' buttons.

2. Enable **Use Personalization**.
3. In **Connector Page Path**, enter the path to the connector JSP page you installed in AEM.
4. If you want your personalized content to run scripts dynamically, enter the path to your JavaScript file in **Script Path**.
5. Click **Save**.

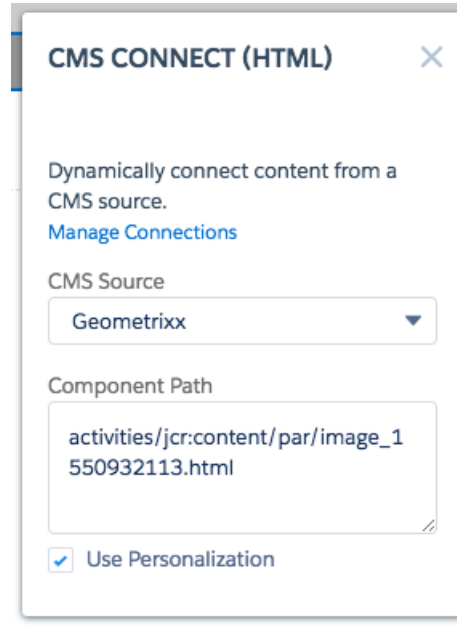
Enable Personalization in Your Page Components

1. Navigate to Community Builder.

- To personalize a header or footer, click  and select **Settings**. Select **CMS Connect**. (Skip this and the next step if you don't want to use a personalized header or footer.)



- In **Header Source** and **Footer Source**, choose the name of the connection that contains the personalized content. In **Header Path** and **Footer Path**, enter the Default Experience URL from AEM for the header and footer components. Append `.html` to default URLs copied from AEM. Enable **Use Personalization** for the header and footer.
- To personalize content on a community page, drag a CMS Connect (HTML) component to your page (or edit an existing one). In the component's property editor, choose the connection name in **CMS Source**. In **Component Path**, paste the Default Experience URL for the component from AEM. Append `.html` to the component path since it's not included in the path in AEM.



5. Voilà! Repeat these steps for any additional components you want to personalize.

[CMS Connector Page Code](#)

If you use CMS Connect to render personalized content in your community, your setup process requires the following connector JSP page code.

CMS Connector Page Code

If you use CMS Connect to render personalized content in your community, your setup process requires the following connector JSP page code.


To get your personalized content from Adobe Experience Manager working in your community, create a JSP connector page that contains the following code. You can add to this code as needed. See [Personalize Your CMS Content](#) for full instructions on setting up personalization in your community.

The connector JSP page logic includes the following sections:

- **Request parameter.** The request parameter (payload) contains the data that a community passes to the connector page. It contains `componentUrls` (an array of component path URLs for which personalization must be run), `asset` (a JavaScript asset specified in the `Asset Path` field in the CMS connection that is injected when the JSP page loads), `clientContext` (IP address, language, country, state, city, latitude, and longitude), `requestId` (a token that is returned as part of the `postMessage` to validate the authenticity of the response), and `domain` (the domain of the community requesting personalized content).
- **Personalization JSP logic.** We provide you with the basic logic, below. You can add logic as needed.
- **JavaScript.** In your JSP, include Salesforce-provided JavaScript that sends a `postMessage` to your community. Construct the script `src` in this way:

```
<your_community_domain/_sfdc/cms-connect/aem_personalization/salesforceConnector.js>
```

Any asset specified in the `Asset Path` field in the CMS connection is included in your JSP.
- **Response.** The final section constructs the response object and does the `postMessage`. The JavaScript that you include in the previous section does this.

 **Note:** To ensure the connector page code gets personalization working in your community, follow these guidelines:

- Don't change any request parameter values.
- Don't take out any try/catch blocks. We need them to handle the case where something goes wrong in the connector page code.
- Don't change the structure of the response object in the `postMessage`.

```
<!-- Salesforce connector to run AEM personalization-->
<%@include file="/libs/foundation/global.jsp"%><%
%><%@page import="
    java.io.StringWriter,
    java.net.URL,
    com.day.cq.wcm.api.WCMMode,
    com.day.cq.wcm.core.stats.PageViewStatistics,
    com.day.text.Text,
    java.util.ResourceBundle,
    com.day.cq.il18n.I18n,
    com.day.cq.personalization.TargetedContentManager,
    com.day.cq.personalization.ClientContextUtil,
    org.apache.sling.commons.json.JSONObject,
    org.apache.sling.commons.json.JSONArray,
    com.day.cq.commons.JS,
    org.apache.sling.engine.*,
    org.apache.sling.api.SlingHttpServletRequest,
    org.apache.sling.api.resource.ResourceResolver,
    java.util.*,
    com.day.cq.*" %><%

%><cq:includeClientLib categories="personalization.kernel"/><%
if(request.getParameter("payload") != null) {
    // For every component URL, get the Teasers object and strategy.
    JSONObject payload = new JSONObject(request.getParameter("payload"));
    JSONArray compUrls = payload.getJSONArray("componentUrls");
    String asset = payload.getString("asset");
    HashMap<String, JSONArray> teaserMap = new HashMap<>();
    HashMap<String, String> strategyMap = new HashMap<>();
    ResourceBundle resourceBundle = slingRequest.getResourceBundle(null);
    I18n i18n = new I18n(resourceBundle);
    final TargetedContentManager targetedContentManager =
sling.getService(TargetedContentManager.class);
    SlingRequestProcessor requestProcessor =
sling.getService(SlingRequestProcessor.class);
    ResourceResolver resolver = slingRequest.getResourceResolver();
    for(int j=0; j<compUrls.length(); j++) {

        JSONArray allTeasers = new JSONArray();
        String strategy = "";

        try {
            String requestPath = new
URL(compUrls.getString(j)).getPath().replaceAll(".html", "");

            Resource resourceObject = resolver.getResource(requestPath);
```

```

Node rootNode = resourceObject.adaptTo(Node.class);
ValueMap prop = resourceObject.adaptTo(ValueMap.class);

// Get the strategy path
String strategyPath = prop.get("strategyPath", (String) null);
if (strategyPath != null) {
    strategy = Text.getName(strategyPath);
    strategy = strategy.replaceAll(".js", "");
}

// Get the campaign path
String campaignPath = prop.get("campaignpath", (String) null);
String campaignClass = "";
if (campaignPath != null) {
    Page campaignPage = pageManager.getPage(campaignPath);
    if (campaignPage != null) {
        campaignClass = "campaign-" + campaignPage.getName();
    }
}

JSONObject teaserInfo =
targetedContentManager.getTeaserInfo(resourceResolver, campaignPath, requestPath);
allTeasers = teaserInfo.getJSONArray("allTeasers");

// Add selectors from the current page for the mobile case, e.g. "smart",
"feature" etc.
String selectors = slingRequest.getRequestPathInfo().getSelectorString();

selectors = selectors != null ? "." + selectors : "";

for (int i = 0; i < allTeasers.length(); i++) {
    JSONObject t = (JSONObject) allTeasers.get(i);
    t.put("url", t.get("path") + "/_jcr_content/par" + selectors + ".html");
}

// Use "default" child node as default teaser and add at the end of the
teaser list
JSONObject defaultTeaser = new JSONObject();
defaultTeaser.put("path", resourceObject.getPath() + "/default");
defaultTeaser.put("url", resourceObject.getPath() + ".default" + selectors
+ ".html");
defaultTeaser.put("name", "default");
defaultTeaser.put("title", i18n.get("Default"));
defaultTeaser.put("campaignName", "");
defaultTeaser.put("thumbnail", resourceObject.getPath() + ".thumb.png");
allTeasers.put(defaultTeaser);
} catch (Exception e) {
    // If an exception occurs for any of the component URLs, we will put default
values in teaserMap and strategyMap
}

teaserMap.put(compUrls.getString(j), allTeasers);
strategyMap.put(compUrls.getString(j), strategy);

```

```

    }

    String requestId = payload.getString("requestId");
    String domain = payload.getString("domain");
    JSONObject teaserJson = new JSONObject(teaserMap);
    JSONObject strategyJson = new JSONObject(strategyMap);

    %>

    <html>
        <head>
            <script type="text/javascript"
src="/etc/clientlibs/granite/jquery.js"></script>
            <script type="text/javascript"
src="/etc/clientlibs/granite/utils.js"></script>
            <script type="text/javascript"
src="/etc/clientlibs/granite/jquery/granite.js"></script>
            <script type="text/javascript"
src="/etc/clientlibs/foundation/jquery.js"></script>
            <script type="text/javascript"
src="/etc/clientlibs/foundation/shared.js"></script>
            <script type="text/javascript"
src="/etc/clientlibs/granite/lodash/modern.js"></script>
            <script type="text/javascript"
src="/etc/clientlibs/foundation/personalization/kernel.js"></script>
            <!--TODO: Include the script that will send a postMessage to your community.
The path of the JS is <your community
domain/_sfdc/cms-connect/aem_personalization/salesforceConnector.js> -->
            <!--The below line injects a script that is passed in the request -->
            <script type="text/javascript" src="<%= asset %>"></script>
        </head>

        <body>
            <script>
                var teaserMap = <%= teaserJson %>;
                var strategyMap = <%= strategyJson %>;
                var requestId = "<%= requestId %>";
                var domain = "<%= domain %>";
                setClientContext();
                var resolvedTeasers = getResolvedTeasers(teaserMap, strategyMap);
                CMS_CONNECT_PERSONALIZATION_AEM.responsePersonalization(resolvedTeasers,
requestId, domain);

                // This is a sample client-context that can be used.
                // Salesforce provides the client-context object which can be used to
build the json
                /**
                 * @typedef {object} payload.clientContext
                 * @property {String} ipAddress User's ipAddress
                 * @property {String} language User language
                 * @property {String} country User's country
                 * @property {String} state User's state

```

```

        * @property {String} city User's city
        * @property {String} latitude User's latitude
        * @property {String} longitude User's longitude
        */
function setClientContext() {
    var payload = <%= payload %>;
    var clientContext = payload.clientContext;
    // Set client-context
    var clientContextJson = {};
    clientContextJson.surferinfo = {
        'IP': clientContext.ipAddress
    };
    clientContextJson.profile = {
        'language': clientContext.language,
        'country': clientContext.country,
        'state': clientContext.state,
        'city': clientContext.city
    };
    clientContextJson.geolocation = {
        'latitude': clientContext.latitude,
        'longitude': clientContext.longitude
    };
    CQ_Analytics.ClientContextMgr.clientcontext = clientContextJson;
}

/**
 * Runs personalization logic for all the component URLs requested
 * @param {teaserMap} Map of componentUrl as key and teasers object
for the component
 * @param {strategyMap} Map of componentUrl as key and strategy for
the component
 */
function getResolvedTeasers(teaserMap, strategyMap) {
    var resolvedTeasers = {};
    for (var key in teaserMap) {
        try {
            if (teaserMap.hasOwnProperty(key)) {
                var resolvedTeaser =
CQ_Analytics.Engine.resolveTeaser(teaserMap[key], strategyMap[key], null);
                resolvedTeasers[key] = resolvedTeaser.url.substring(1);
            }
        } catch (err) {
            // If any error occurs in calculating resolved teaser for
a component url, we save it as error
            resolvedTeasers[key] = "error";
        }
    }

    return resolvedTeasers;
}
</script>
</body>
</html>

```

```
    <%  
  }  
  %>
```

CMS Connect Recommendations for Optimal Usage

Read these tips and gotchas to get the most out of CMS Connect.

Scope Your CSS

Your Salesforce community pages can have CSS. Your CMS connections can have CSS. To avoid rule collision on your community's pages, we recommend scoping your CSS.

Scoping involves adding a `DIV` class in the code to "tag the tags," which prevents rule collision by marking the CSS from your CMS with a prefix so that it's given a higher priority.

For example, your community page specifies 10 point font, while your CSS has 14 point font. Use a scope prefix on your CSS to determine which rule gets priority.

Minify and reminify your CSS

The downside of scoping your CSS is that it increases your code's file size by 10 to 20%, which translates to longer download time for your viewers. But you can more than make up for this performance hit by minifying and reminifying your code. Plan to include this work as part of your build time for your CMS website. It's worth doing so you can reap the benefits of scoping without degrading performance.

CSS should use REM at 100%

If the content on your pages looks too big, it's possible that your CSS is using REM with the old technique of 62.5%. The root page of Salesforce uses REM at 100%. Recode your CSS at 100%.

Include only relevant CSS and JavaScript

Parsing CSS and JavaScript files takes time. For optimal performance on your community pages, link only to CSS and script files that have been tailored for the pages you plan to display them on. Your efforts to plan ahead will be rewarded in faster load times for your audience viewing the content.

Serve JavaScript libraries with initialization

You can use JavaScript for content such as a carousel or a menu system on your community pages. But make sure that this JavaScript runs after the HTML loads on a page and not before.

Typically, you define the libraries (like `jQuery` and `jQuery` plugins such as a carousel) as part of the CMS Connect configuration to make sure that they load early, that they are always present on the page, and that they are ready to be used by multiple fragments. Include the initialization code specific to each HTML fragment (the JavaScript that created the instance of a carousel, for example) in a script tag at the bottom of that fragment.

Don't include fragment-specific initialization code in the JavaScript files of your CMS Connect configuration because those files are executed as early as possible to emulate head scripts, and the page body won't be ready. Instead, make the initialization code part of

your HTML, much like local JavaScript is part of the Lightning component. You might need to adjust your existing code because sometimes the initialization code of all widgets on a page is grouped together or placed in a different location on the page.

CMS Connect Examples

Here are some examples of how to set up CMS Connect in your community.

[Example: Connect JSON Content to Your Community](#)

Here's an example of how to set up JSON content in your community using CMS Connect. This example connects your community to a WordPress blog with JSON content.

Example: Connect JSON Content to Your Community


Here's an example of how to set up JSON content in your community using CMS Connect. This example connects your community to a WordPress blog with JSON content.



Example: You have a Wordpress blog called Capricorn Cafe. You'd like to connect it to your Salesforce community using CMS Connect. Here are example endpoints for a Content List and Content Item:

- Content List:
`https://public-api.wordpress.com/rest/v1.1/sites/capricorncafeblog.wordpress.com/posts?number=6&page=1`
- Content Item:
`https://public-api.wordpress.com/rest/v1.1/sites/capricorncafeblog.wordpress.com/posts/38`

Set up JSON in your CMS connection:

1. Follow the steps to [create a CMS connection](#) on page 56.
 2. For the server URL, enter `https://public-api.wordpress.com/`.
 3. You don't need to enter a root path.
 4. In the JSON section, click **Add JSON**.
 5. Enter the following to add a list of blog posts:
 - Name: *Blog List*
 - Type: **Content List**
 - Path:
`rest/v1.1/sites/capricorncafeblog.wordpress.com/posts?number={itemsPerPage}&page={pageNumber}`
-  **Note:** The path has 2 variables used for pagination. `{itemsPerPage}` is the number of items to be displayed on a page, with the value to be computed dynamically during rendering of the CMS Connect (JSON) component. `{pageNumber}` is the current page number, whose value is to be computed dynamically during rendering of the CMS Connect (JSON) component.
6. Click **Add JSON**.
 7. Enter the following to add a single blog post:
 - Name: *Blog Item*
 - Type: **Content Item**
 - Path: `rest/v1.1/sites/capricorncafeblog.wordpress.com/posts/{component}`



Note: The `{component}` variable adds the component path dynamically, based on what's entered for Component Path in the property editor for the CMS Connect (JSON) component in the Builder.

8. Click **Save**.

Add the blog post to a page

1. Create a new standard community page, using the "New Page" option.
2. In the Builder, drag a **CMS Connect (JSON)** component on a page.
3. In the property editor, select the CMS source that contains your JSON content.
4. Select the name of your JSON content item (blog post), in this case **Blog Item**.
5. For the component path, enter `{!id}`.
6. For the content item layout, select **Detail**.
7. Enter the following expressions in the Title, Author, Published On, Body, and Image Source fields:

- `@title`
- `@author/name`
- `@date`
- `@content`
- `@featured_image`

8. Click **Save** in the property editor.

Add the list of blogs to a page

1. In the Builder, drag a **CMS Connect (JSON)** component on a page.
2. In the property editor, select the CMS source that contains your JSON content.
3. Select the name of your JSON content list, in this case **Blog List**.
4. For the content list node path, enter `@posts`.
5. For content list layout, select **Grid**.
6. For items per page, enter `5`.
7. For columns, enter `2`.
8. For content list item layout, select **Card**.
9. Enter the following expressions in the Title, Author, Published On, Body, and Image Source fields:

- `@title`
- `@author/name`
- `@date`
- `@content`
- `@featured_image`

10. In the Navigation Link section, enter `Read More` for the link text.

11. For the type, select **My Pages**.

12. For the page, select the name of the page that contains the content item (blog post).

13. In the URL Parameter Mapping section, enter `id` for the name, and `@ID` for the value. (If you don't see the Name and Value fields, click **Add Parameter**.)
14. Click **Save** in the property editor.

CHAPTER 8 Community Migration, Packaging, and Distribution

In this chapter ...

- [Migrate Your Community with Change Sets](#)
- [Lightning Bolt Solutions: Build Once, Then Distribute and Reuse](#)

You can migrate your community between related orgs, such as your sandbox and production org, using change sets. And with managed packages, you can distribute customized Lightning Bolt solutions or pages to other Salesforce users and orgs, including people outside your company.

EDITIONS

Available in: Salesforce Classic and Lightning Experience

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

Migrate Your Community with Change Sets

Use change sets to move your community between related orgs that have a deployment connection, such as your sandbox and production orgs. Create, customize, and test your community in your test environment and then migrate the community to production when testing is complete.

You can use change sets to move Lightning communities and Salesforce Tabs + Visualforce communities.

1. Create and test your community in your preferred test org, such as sandbox.
2. From Setup in your test org, enter *Outbound Change Sets* in the Quick Find box, and then select **Outbound Change Sets**.
3. Create a change set, and click **Add** in the Change Set Components section.
4. Select the **Network** component type, choose your community, and then click **Add to Change Set**.
5. To add dependent items, click **View/Add Dependencies**. We recommend selecting all the dependencies listed.



Tip:

- For navigation menus that link to standard objects, custom list views aren't included as dependencies. Manually add the custom list view to your change list.
- Manually add new or modified profiles or permission sets referenced in **Administration > Members**.
- The list of dependencies has two Site.com items—*MyCommunityName* and *MyCommunityName1*. *MyCommunityName* holds the various Visualforce pages that you can set in Administration (in Community Workspaces or Community Management). *MyCommunityName1* includes the pages from Community Builder.

6. Click **Upload** and select your target org, such as production.
Make sure that the target org allows inbound connections. The inbound and outbound orgs must have a deployment connection.
7. In your target org, create a community (if one doesn't exist) with the same name and template version as the community in your source org.
For Communities, you can make updates only with change sets, which means that you can't create a community directly from an inbound change set.
8. From Setup, select **Inbound Change Sets** and find the change set that you uploaded from your source org.
9. Validate and deploy the change set to make it available in the target org.



Warning: When you deploy an inbound change set, it overwrites the community in the target org.

10. Manually reconfigure any [unsupported items](#) in the target org community.

EDITIONS

Available in: Salesforce Classic and Lightning Experience

Available in: **Enterprise, Performance, Unlimited,** and **Developer** Editions

USER PERMISSIONS

To customize or publish a community:

- Create and Set Up Communities

To edit deployment connections and use inbound change sets:

- Deploy Change Sets AND Modify All Data

To use outbound change sets:

- Create and Upload Change Sets, Create AppExchange Packages, AND Upload AppExchange Packages

11. Add data for your community, and test it to make sure that everything works as expected. Then publish your changes to go live.

SEE ALSO:

- [Considerations for Migrating Communities with Change Sets](#)
- [Change Sets Best Practices](#)
- [Upload Outbound Change Sets](#)
- [Deploy Inbound Change Sets](#)

Considerations for Migrating Communities with Change Sets

Keep the following considerations and limitations in mind when migrating your Lightning or Salesforce Tabs + Visualforce community with change sets.

EDITIONS

Available in: Salesforce Classic and Lightning Experience

Available in: **Enterprise, Performance, Unlimited, and Developer** Editions

General

- For Communities, you can make updates only with change sets, which means that you can't create a community directly from an inbound change set. Instead, in the target org, create a community with the same name and template version, and then deploy the inbound change set.
- When you deploy an inbound change set, it overwrites the community in the target org. So although you can't use a change set to delete a component, such as a community, you can delete the pages within a Lightning community. For example, let's say you delete pages from a Lightning community in sandbox and then create an updated outbound change set. When you redeploy the change set in a target org, such as production, the pages are also deleted there.
- If you [update the community template](#) in the source org to unify its Community Builder branding properties, ensure that the template is also updated in the target org before deploying the change set.

Administration

Administration settings are in Community Workspaces or Community Management.

- Remember to add any new or modified profiles or permission sets referenced in **Administration > Members** to your outbound change set. They're not automatically included as dependencies.
- For communities created in a sandbox org before the Summer '17 release, you must resave administration settings prior to migration to transfer them successfully.
- Until you publish your community in the target org, settings for the change password, forgot password, home, self-registration, and login pages appear to return to their default values.
- To update settings in the Members area and the Login & Registration area, you must deploy the changes in separate change sets. First update and deploy the Members area setting, and then update and deploy the Login & Registration settings.

Navigation Menu

The Navigation Menu component is available in Community Builder for Lightning communities.

- For menu items that link to objects, list views are reset to the default list view. Also, custom list views for standard objects aren't included as dependencies.
- Until you publish you community in the target org, menu items that point to community pages appear to be broken.

Recommendations

- Updates to recommendation names aren't supported. If you change the name of a recommendation in the source org having previously migrated it, the target org treats it as a new recommendation.
- Recommendation images aren't supported.
- When you deploy an inbound change set, it overwrites the target org's scheduled recommendations with those from the source org.

Unsupported Settings and Features

The following items aren't supported. Manually add them after you deploy the inbound change set.

- Navigational and featured topics
- Audience targeting
- Branding sets
- Languages
- Dashboards and engagement
- CMS Connect
- Recommendation images
- Branding panel images in Community Builder
- The following Administration settings in Community Workspaces or Community Management:
 - The Account field in the Registration section of the Login and Registration area
 - The Settings area
 - The Rich Publisher Apps area

SEE ALSO:

[Change Sets Best Practices](#)

[Change Sets Implementation Tips](#)

[Migrate Your Community with Change Sets](#)

Lightning Bolt Solutions: Build Once, Then Distribute and Reuse

With Lightning Bolt, you can create and export industry-specific solutions to jump-start new communities or package and distribute them for others to use. Save time by building once and then reusing. Whether it's for your own org or you're a consulting partner or ISV, you can reduce the time required to build communities and cut development costs.



EDITIONS




Available in: Salesforce Classic and Lightning Experience

Available in: **Enterprise, Performance, Unlimited,** and **Developer** Editions

A Lightning Bolt solution comprises a customized Lightning template that's made up of a theme layout and CSS, along with pages, content layouts, and Lightning components. In addition, Lightning Bolt solutions seamlessly integrate with Salesforce and incorporate business logic, custom objects and apps, industry best practices, and more.

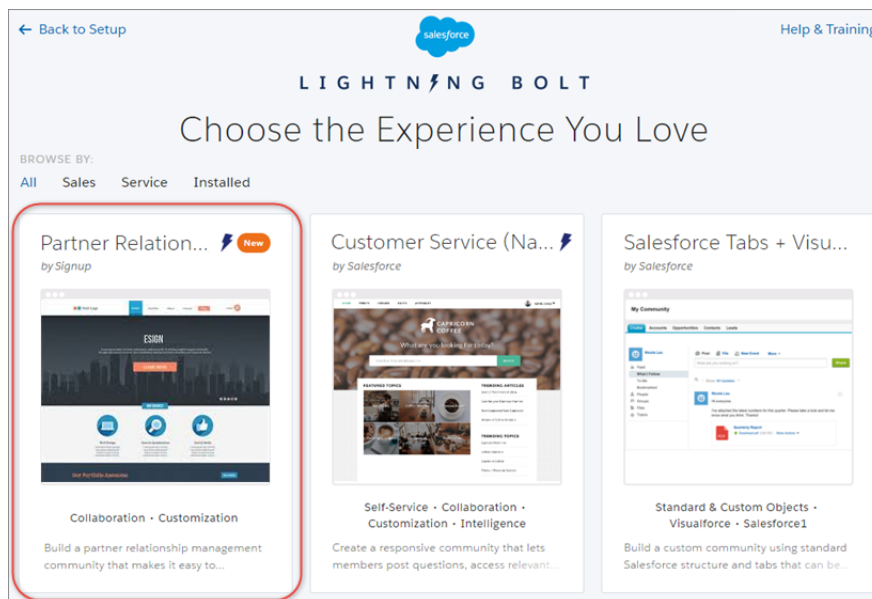
Use any Lightning template as a base to build your custom solution with standard pages and components, or create custom pages, layouts, and components of your own. After you finish customizing the template, export the template or one of its pages from **Settings > Developer** to make it ready to use or distribute.

You can take advantage of Lightning Bolt in several ways to save time and money. Here are just a few examples.

-  **Example:** You've customized the Partner Central template to create a community with features that meet your particular business needs. You want to create several similar partner communities, but you don't want to build each one from scratch. By exporting your customized solution, you can reuse it to build as many communities as you need.
-  **Example:** You're a consulting partner who specializes in building communities for the real estate industry. With Lightning Bolt, you can build and export a real-estate-focused solution that's easy to distribute to your customers. After the template is installed on a customer's org, you can further customize it to suit their unique needs. By building the bulk of the solution in your org and then distributing it to your customers' orgs, you can launch their communities in no time.
-  **Example:** You're an ISV who builds a custom page and several custom Lightning components to create an e-commerce feature for use in Community Builder. Now you can bundle the page and its components into a single package and distribute it to your customers.

Reuse Your Own Solution

When you export a template, it appears in the Community Creation wizard in your org, where you can use it to build new communities.



Similarly, you can export a single page, which includes the page's content layout and components. After you export a page, it appears in the New Page dialog box in all communities in your org.

Package and Distribute Solutions

You can package solutions for distribution to your customers' orgs. After you create and upload a managed package, share the link privately with your clients, customers, or partners. Alternatively, publish your custom Lightning Bolt solution as a managed package to AppExchange. Market your solution with an AppExchange listing in the same way you list any other app, component, or consulting service. Describe your solution, pricing, support, and other details so that customers can determine whether your offering is right for them.

When a template is installed from another org, it appears in the org's Community Creation wizard. Installed pages appear in the New Page dialog box.

SEE ALSO:

[Salesforce Partner Community: Lightning Bolt for Partners](#)


[Lightning Component Developer Guide: Create Custom Theme Layout Components for Communities](#)

[ISVforce Guide: Creating and Uploading a Managed Package](#)

[Package and Distribute Your Apps](#)

Export and Packaging Considerations for Lightning Bolt Solutions

Before you export a customized template or page as a Lightning Bolt solution, keep the following considerations and limitations in mind.

 **Tip:** We recommend using managed packages to avoid naming conflicts with other packages in your customer's org or your own.

Template Export and Packaging

- You can export any customized Lightning template as a Lightning Bolt solution, other than Koa and Kokua, which are being retired.
- The exported template name must be unique.
- When you export a template, the system removes non-alphanumeric characters from the template and page names. For example, *My Template #2* becomes *My_Template_2*.
- In the Community Creation wizard, the template author (for example, *by Salesforce*) for exported and imported templates differs. When you export a template, your org name is shown within your own org. When you import a template, the publisher name of the package is shown.

EDITIONS

Available in: Salesforce Classic and Lightning Experience

Available in: **Enterprise, Performance, Unlimited, and Developer** Editions

- For navigation menu items that link to objects, list views are reset to the default list view. Also, custom list views for standard objects aren't included as dependencies, although custom list views for custom objects are.
- Only the following Administration settings (in Workspaces or Community Management) are included.
 - Change password page
 - Forgot password page
 - Home page
 - Login page
 - **Allow internal users to log in directly to the community** option
- The following items are not included when you export a template. After you import the template and use it to create a community in the destination org, you must manually reconfigure these items.
 - Most Administration settings (except for the settings already listed).
 - Community Builder settings, including head markup and the Google Tracking ID.
 - Custom theme layouts that aren't in use. Only theme layouts that are selected in **Settings > Theme** are included.
 - Custom styles in the CSS editor.
 - Localized content for multilingual communities.
 - Non-default page variations; only default page variations are included. If a page doesn't have a default variation—for example, a page with two variations that are both set to audience-based visibility—the page is excluded entirely.
- If you upgrade a managed template package, existing communities that are based on the upgraded template aren't updated.

Single Page Export and Packaging

- The exported page name must be unique.
- When you export a page, the system generates a developer name (`devName`) for it by prepending the community name and removing non-alphanumeric characters. For example, *My #awesome page* in the Acme community becomes `Acme_My_awesome_page`. Developer names longer than 80 characters are truncated.
- Exported page variations use the naming convention `[Community Name]_[Page Name]_[Variation Name]`.

Original Page Name	Exported Page Name	Exported Page Developer Name
Coffee Fans	Coffee Fans	Acme_Coffee_Fans
West Coast (page variation)	Coffee Fans - West Coast	Acme_Coffee_Fans_West_Coast
East Coast (page variation)	Coffee Fans - East Coast	Acme_Coffee_Fans_East_Coast

- For images in the Rich Content Editor, we export the version used in the editor, which is not necessarily the latest version of the asset file.
- Audience-based visibility criteria aren't included in the export process. Manually reenter this information after importing the page to the community in the destination org.
- When you export a page, its page variations aren't included in the package. You must export them separately.

- You can't export individual login pages.

SEE ALSO:

[Export a Customized Lightning Bolt Solution](#)

[Export a Customized Lightning Bolt Page](#)

[Package and Distribute Your Apps](#)

[ISVforce Guide: Creating and Uploading a Managed Package](#)

Requirements for Distributing Lightning Bolt Solutions

Whether you're a partner, ISV, or developer, before Salesforce can recognize a customized template as a Lightning Bolt solution that's ready for distribution on AppExchange, your solution must meet certain requirements.

Use this checklist to ensure that you have:

- Developed a custom solution using a Lightning template (apart from Koa and Kokua)
- Included at least one custom Lightning component (API version 40.0 or later)
- Included at least one custom Theme Layout component with a unique visual design
- Tested with the "Stricter Content Security Policy for Lightning Components in Communities" critical update enabled in sandbox
- Used the Salesforce Lightning CLI plug-in to test custom Lightning components
- Tested all community functionality and appearance across desktop, tablet, and mobile devices
- Ensured that your customizations have no regressions (each release)
- Verified that all third-party technology you use is approved by Salesforce
- Ensured proper code coverage, and executed basic performance testing
- Adhered to all Salesforce Lightning developer guidelines (such as attribute enforcements)
- Provided installation and configuration documentation
- Provided appropriate customer support for custom functionality

EDITIONS

Available in: Salesforce Classic and Lightning Experience

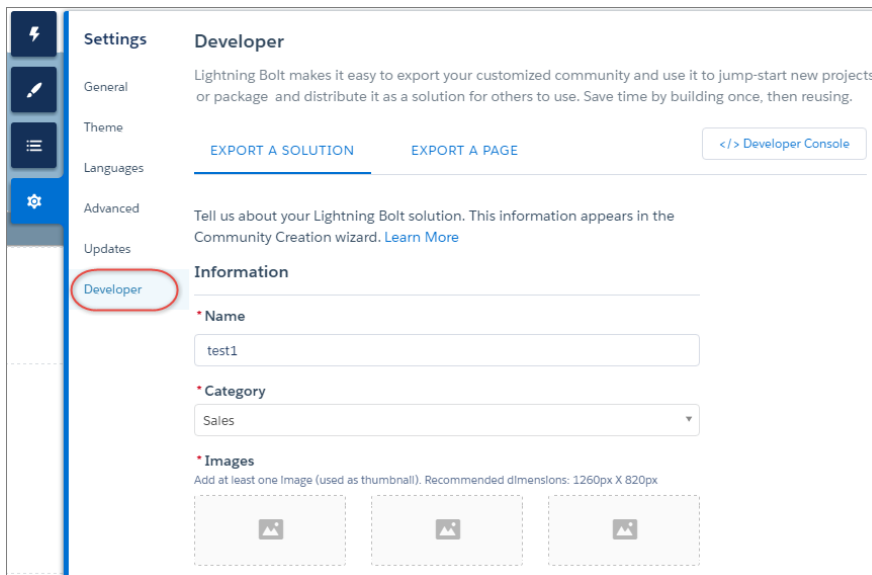
Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

Export a Customized Lightning Bolt Solution

With Lightning Bolt, you can customize and export the Customer Service (Napili) template to use as a base for your new communities, or package and distribute the solution for others to use.

 **Note:** You can't export a Lightning Bolt solution until you [update the template](#) to unify its branding properties.

1. In Community Builder, select **Settings > Developer**.
The information that you add on this page appears in the Community Creation wizard and helps users understand the purpose and benefits of your solution.
2. Add a unique name for the solution, and select a category.



3. Add at least one image to use as the thumbnail image. The recommended image dimensions are 1260 x 820px. You can add two more images that appear in the detailed description of the solution.
4. Enter a summary that describes the purpose of the solution.
5. Enter at least one key feature.
The feature titles appear under the thumbnail in the Community Creation wizard. The feature descriptions appear in the solution's detailed description.
6. Click **Export**.
After you export the solution, it appears in the Community Creation wizard in your org. You can then use it as a base for building new communities. If you package the solution and install it in another org, it appears in that org's Community Creation wizard.

EDITIONS

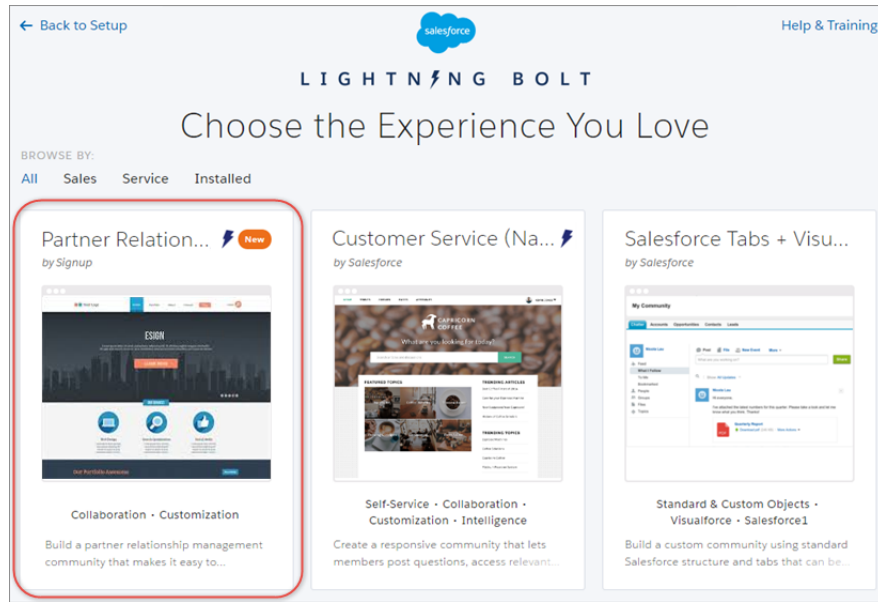
Available in: Salesforce Classic and Lightning Experience

Available in: **Enterprise, Performance, Unlimited,** and **Developer** Editions

USER PERMISSIONS

To create, customize, or publish a community:

- Create and Set Up Communities AND View Setup and Configuration



- To distribute the Lightning Bolt solution for others to use, [create a managed package](#).

 **Note:** To delete an exported or imported solution, from Setup, enter *Lightning Bolt Solutions* in the Quick Find box, and then click **Lightning Bolt Solutions**.

Deleting a solution from your org doesn't affect communities that are already based on it.

SEE ALSO:

[Export and Packaging Considerations for Lightning Bolt Solutions](#)

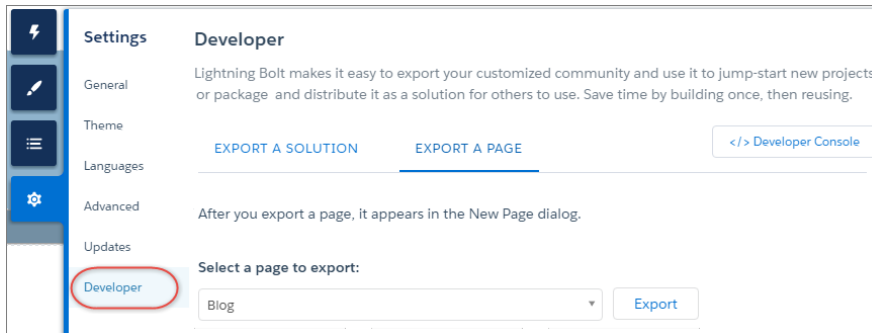
[Package and Distribute Your Apps](#)

[ISVforce Guide: Creating and Uploading a Managed Package](#)

Export a Customized Lightning Bolt Page

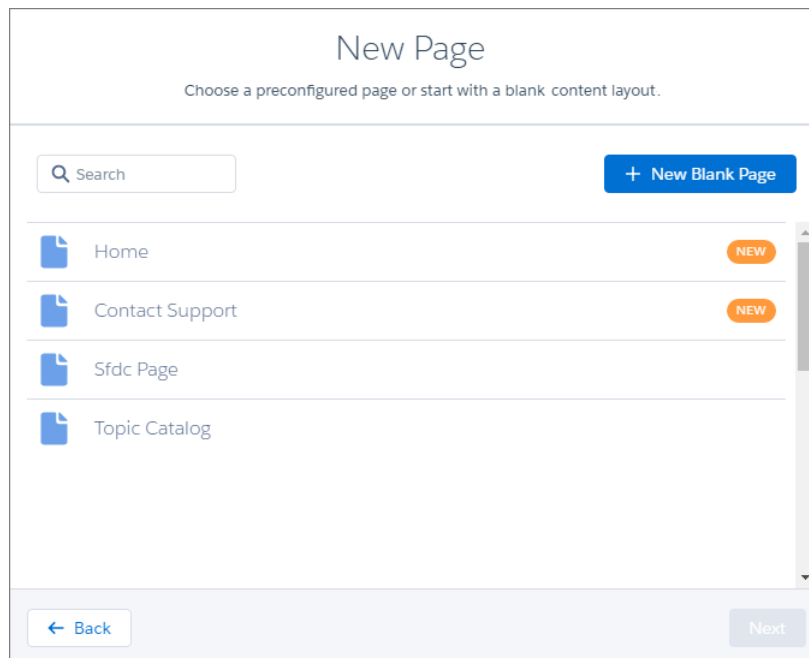
Export pages that you've customized in a Lightning template to use as a base for new pages, or package and distribute them for others to use.

1. In Community Builder, select **Settings > Developer** and click **Export a Page**.



2. Select the page to export.
3. Click **Export**.

After you export a page, it appears in the New Page dialog box in all the communities in your org. If you package the page and install it in another org, it appears in the New Page dialog box of that org. Newly installed pages are highlighted as New for 30 days.



4. To distribute your customized page for others to use, [create a managed package](#).

EDITIONS


Available in: Salesforce Classic and Lightning Experience

Available in: **Enterprise, Performance, Unlimited,** and **Developer** Editions

USER PERMISSIONS

To create, customize, or publish a community:

- Create and Set Up Communities AND View Setup and Configuration

 **Note:** To delete an imported or exported page, from Setup, enter *Lightning Bolt Pages* in the Quick Find box, and then click **Lightning Bolt Pages**.

Existing pages in Community Builder that are based on the deleted page are unaffected. However, deleted pages no longer appear in the New Page dialog.

SEE ALSO:

[Export and Packaging Considerations for Lightning Bolt Solutions](#)

[Package and Distribute Your Apps](#)

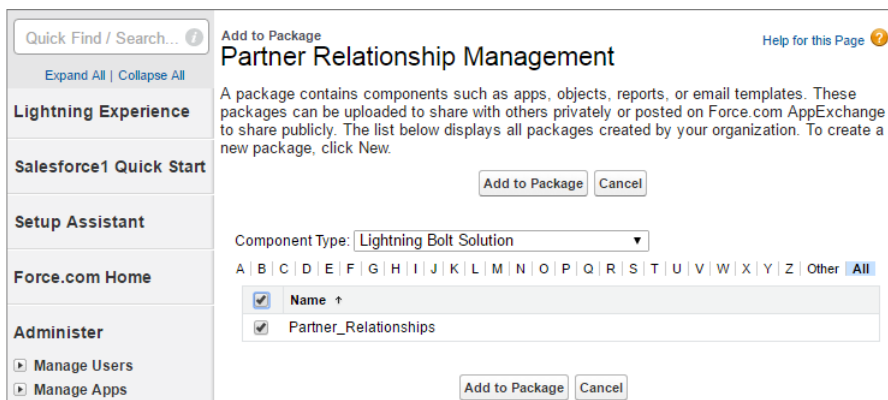
[ISVforce Guide: Creating and Uploading a Managed Package](#)

Package and Distribute Lightning Bolt Solutions or Pages

After you export a Lightning Bolt solution or a page, you can create a managed package to distribute it to other users or orgs, including those outside your company. You can also upload solutions for distribution on AppExchange.

A package is a container for something as small as an individual component or as large as a set of related apps. Packages come in two forms—unmanaged and managed—but we recommend using only managed packages to avoid naming conflicts with other packages in your customer's org or your own. To create a managed package, use a Developer Edition org.

1. From Setup, enter *Package* in the Quick Find box, and then click **Packages**.
2. To package an exported Lightning Bolt solution, select **Lightning Bolt Solution** as the component type and add your solution to the package. To package an exported page, select **Lightning Page** as the component type. All supported dependencies are included.



3. Upload the package. Then distribute it on AppExchange, or share the link privately with your clients, customers, or partners.

SEE ALSO:

[ISVforce Guide: Overview of Packages](#)

[ISVforce Guide: Creating and Uploading a Managed Package](#)

[ISVforce Guide: Publish Your Offering on the AppExchange](#)

EDITIONS

Available in: Salesforce Classic and Lightning Experience

Available in: **Enterprise, Performance, Unlimited,** and **Developer** Editions

USER PERMISSIONS

To create, customize, or publish a community:

- Create and Set Up Communities AND View Setup and Configuration

To create and upload packages:

- Create AppExchange Packages AND Upload AppExchange Packages

INDEX

A

Adobe Experience Manager [54, 56, 62](#)
AEM Personalization [67](#)

B

Branding panel [14–15](#)

C

Client Context [67](#)
CMS [54, 56, 58–59, 62](#)
CMS components [63](#)
CMS Connect [54–56, 58, 62, 75](#)
CMS Connect (HTML) [63](#)
CMS Connect (JSON) [63](#)
CMS Connect examples [76](#)
CMS Connect optimal usage [75](#)
CMS Connect personalization [70](#)
CMS Connect prerequisites [55](#)
Communities
 AppExchange [79](#)
 change sets [80–81](#)
 distribute [90](#)
 distribution [79](#)
 Lightning Bolt solution [79, 90](#)
 migrate [81](#)
 migration [79](#)
 package [90](#)
 packaging [79](#)
Community Builder
 configure components [7](#)
 content layouts [28](#)
 distribution requirements [86](#)
 export a page [89](#)
 export a template [87](#)
 Lightning Bolt solution [82, 84, 86–87, 89](#)
 packaging [82, 87, 89](#)
 packaging considerations for Lightning Bolt solutions [84](#)
 page export [82, 84, 89](#)
 template export [82, 84, 87](#)
component
 theme layout [25](#)
component bundles
 configuration tips [11](#)
 design resources [10](#)
component paths [58](#)

components
 profile menu [29](#)
 search [29](#)
configure components [7](#)
configure theme layout component [25](#)
connection load order [62](#)
connector page [70](#)
content layout component [14](#)
Content Security Policy (CSP) [45–46](#)
critical update [46](#)
CSS Editor [14–15, 21](#)
custom content layouts
 creating for Community Builder [28](#)
custom CSS [15](#)
custom font [21](#)
Customer Service (Napili) [5](#)

D

design resources [10](#)
Developer Console [7](#)
disable CMS connection [62](#)
Drupal [54, 56, 62](#)

E

enable CMS connection [62](#)
events [12](#)
example
 basic structure [35](#)
 logo [37](#)
 navigation menu [38](#)
 properties [43](#)
 search [39](#)
 theme layout component [34](#)
 token bundle [37](#)

F

font [21](#)
footer [63](#)

H

header [63](#)
HTML [64](#)

I

interfaces [12](#)
introduction [1–3](#)

Index

J

JSON [59](#)
JSON content [64, 76](#)
JSP [70](#)

L

language mapping [58](#)
Lightning Bolt solution [79, 82, 84, 86–87, 89–90](#)
Lightning Component framework [3](#)
Lightning templates [5](#)
LockerService [45–46](#)
logo [37](#)

N

navigation menu [38](#)

P

Partner Central [5](#)
Personalization [67](#)
profile menu [29](#)

R

resources [2](#)

root path [58](#)

S

Salesforce Lightning [3](#)
Salesforce Lightning Design System [31](#)
SDL [54, 56, 62](#)
search [29, 39](#)
security [45–46](#)
Sitecore [54, 56, 62](#)
SLDS [31](#)
standard design tokens [31](#)
stricter CSP [45–46](#)

T

theme layout component [14, 23, 25, 34–35, 37–39, 43](#)
theme layout type [23](#)
token bundle [37](#)

W

WordPress [54, 56, 62](#)
Wordpress blog [76](#)