Lightning Communities Developer Guide

Version 44.0, Winter '19

@salesforcedocs

Last updated: December 4, 2018

CONTENTS

CHAPTER 1 Get Up to Speed with Lightning Communities

In this chapter ...

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 Lightning 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 template or a Lightning Bolt Solution
- Experience using Community Builder and the Customer Service 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 (Help)
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.

Connect Your Community to Your Content Management System (Guide)
Use CMS Connect to embed content from a third-party content management system (CMS), such as Adobe Experience Manager (AEM) or Drupal, 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 and Partner Central community templates
- Lightning Bolt Solutions because they're based on the Customer Service 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 template.

Customer Service

The Customer Service 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 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?

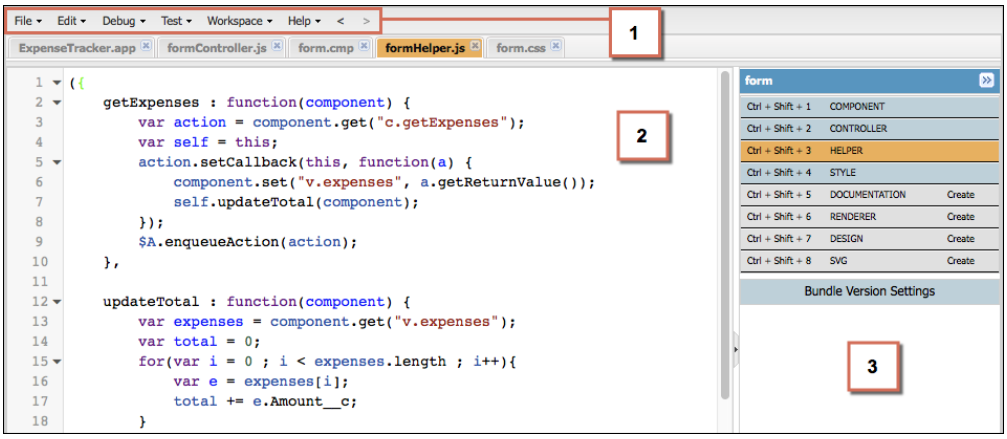
CHAPTER 2 Develop Lightning Communities: The Basics

In this chapter ...

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.




```
({
    getExpenses : function(component) {
        var action = component.get("c.getExpenses");
        var self = this;
        action.setCallback(this, function(a) {
            component.set("v.expenses", a.getReturnValue());
            self.updateTotal(component);
        });
        $A.enqueueAction(action);
    },

    updateTotal : function(component) {
        var expenses = component.get("v.expenses");
        var total = 0;
        for(var i = 0 ; i < expenses.length ; i++){
            var e = expenses[i];
            total += e.Amount__c;
        }
```

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

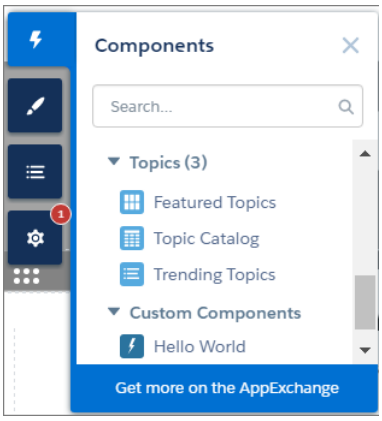
Configure Drag-and-Drop Components for Community Builder

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: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

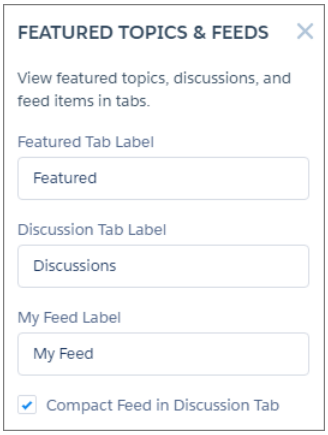
Exposing Component Attributes in Community Builder

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.

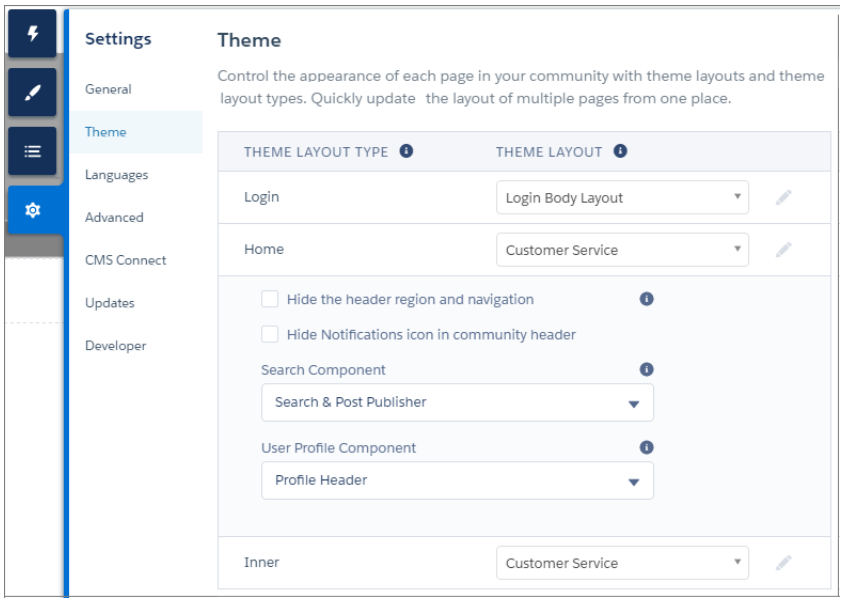
To make a Lightning component attribute available for administrators to edit in Community Builder, add a `design:attribute` node for the attribute in the design resource. When you mark an attribute as required, it automatically appears in Community Builder, unless it has a default value assigned to it.

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

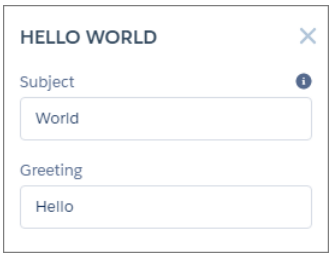
Keep these guidelines in mind when creating components and component bundles for Community Builder.

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 Locker Service 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.



- 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
- force:HasRecordId
- lightningcommunity:allowInRelaxedCSP

Components

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

Events

- force:createRecord
- force:editRecord
- force:navigateToSObject
- force:navigateToList
- force:navigateToRelatedList
- force:navigateToURL
- force:navigateToObjectHome
- force:refreshView
- force:showToast (not available on login pages)
- forceCommunity:analyticsInteraction
- forceCommunity:routeChange
- forceCommunity:setActiveLanguage
- lightning:openFiles
- lightningcommunity:deflectionSignal

SEE ALSO:

Component Reference

Interface Reference

Event Reference

CHAPTER 3 Customize the Look and Feel of a Lightning Template

In this chapter ...

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 Theme panel updates the template with simple, point-and-click properties. This approach is ideal for admins to use.

  Note: Individual component property panels, such as for headers or search, provide additional more specific adjustments in look and feel.

- 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.

  Note: With the introduction of theme swapping, any custom CSS you have entered is now tied directly to your active theme. For existing communities, ensure that any desired custom CSS you may currently be using is copied to the newly selected theme. Where custom CSS is necessary, leverage public Lightning Communities tokens in your theme layouts where it makes sense (`brandLogoImage`, `action color`, etc) to ease the pain of future updates to the template and themes.

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

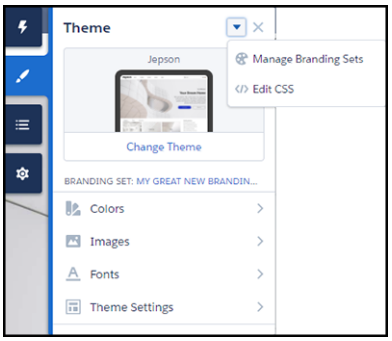
- Custom Lightning components encapsulate a CSS resource as part of the component bundle, making the components reusable across communities.
- Content layout components define the content regions of your page and contain components.
- Theme layout components let you customize the structural layout of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

Update a Template with the Theme Panel

Within Community Builder, the simplest way to change the look of a template is with the Theme panel. Administrators can quickly style an entire community using the theme panels to apply colors, specify fonts, add a logo, or adjust general page structure and defaults.

The properties set in the Theme panel apply to the pages in a template and most off-the-shelf components. Use Branding Sets to quickly apply bundles of colors, images, and fonts.

The Theme panel's properties also apply to custom Lightning components that use standard design tokens to control their appearance.



Note: If you are still using the template from Spring '17, to unify the theme properties of the login pages with the rest of the pages of your community, update your template in **Settings** > **Updates**. Otherwise, you have to theme the login pages separately.

SEE ALSO:

Salesforce Help: Update Your Community's Template

Salesforce Help: Theme Your Community with Community Builder

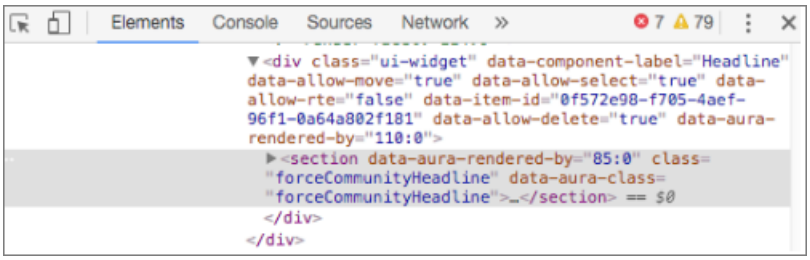
Override Template Elements with Custom CSS

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 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.

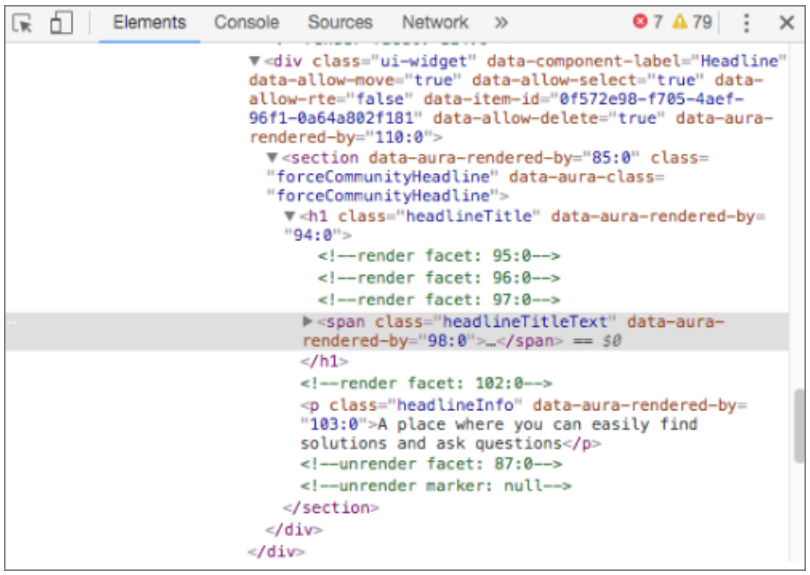
To make minor CSS modifications to a template item, use Chrome DevTools to inspect the page and discover the item's fully qualified name and CSS class. Then use the information to override the item's standard CSS with your custom CSS. To learn more about inspecting and editing pages and styles, refer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

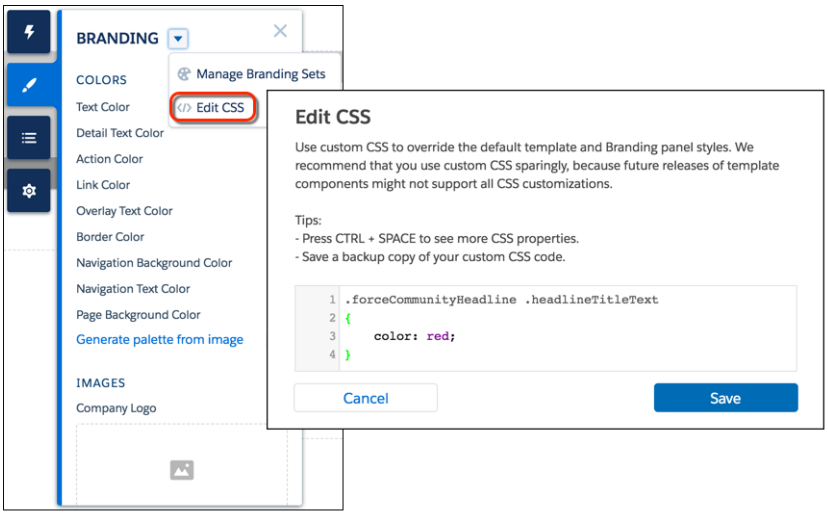
Then find the element that you want to style—for example, `headlineTitleText`. If the element doesn't have a class name, you must write a specific selector targeting the element.



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 Lightning communities update. 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 Lightning communities update. If you plan to continue using custom CSS overrides, migrate them forward after you update the template.

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.

See each component's documentation for a detailed list of all selectors.

- Navigation Menu on page 19
- Panels Container on page 22
- Record Banner on page 23
- Record Detail on page 23
- Record Layout on page 24
- Record List on page 26
- Record Related List on page 27
- Related Articles on page 27
- Reputation Leaderboard on page 28
- Trending Articles by Topic on page 30

CSS Overrides Migration for the Navigation Menu

Many CSS selectors for the Navigation Menu changed with the Lightning communities 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.

CSS Overrides Migration for the Panels Container

Some CSS selectors for the Panels Container changed with the Lightning communities 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.

CSS Overrides Migration for the Record Banner Component

Some CSS selectors for the Record Banner component changed with the Lightning communities update. If you plan to continue using custom CSS overrides, migrate them forward after you update the template.

CSS Overrides Migration for the Record Detail Component

Some CSS selectors for the Record Detail component changed with the Lightning communities update. If you plan to continue using custom CSS overrides, migrate them forward after you update the template.

CSS Overrides Migration for the Record Layout Component

Some CSS selectors for the Record Layout component changed with the Lightning communities update. If you plan to continue using custom CSS overrides, migrate them forward after you update the template.

CSS Overrides Migration for the Record List Component

Some CSS selectors for the Record List component changed with the Lightning communities update. If you plan to continue using custom CSS overrides, migrate them forward after you update the template.

CSS Overrides Migration for the Record Related List Component

Some CSS selectors for the Record Related List component changed with the Lightning communities update. If you plan to continue using custom CSS overrides, migrate them forward after you update the template.

CSS Overrides Migration for the Related Articles Component

Some CSS selectors for the Related Articles component changed with the Lightning communities update. If you plan to continue using custom CSS overrides, migrate them forward after you update the template.

CSS Overrides Migration for the Reputation Leaderboard Component

Some CSS selectors for the Reputation Leaderboard component changed with the Lightning communities update. If you plan to continue using custom CSS overrides, migrate them forward after you update the template.

CSS Overrides Migration for Trending Articles by Topic Component

Some CSS selectors for the Trending Articles by Topic component changed with the Lightning communities update. If you plan to continue using custom CSS overrides, migrate them forward after you update the template.

SEE ALSO:

Override Template Elements with Custom CSS

CSS Overrides Migration for the Navigation Menu

Many CSS selectors for the Navigation Menu changed with the Lightning communities 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 for the Navigation Menu.

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

`.forceCommunityNavigationMenu`
`.forceCommunityNavigationMenu #navigationMenu`
`.forceCommunityNavigationMenu .navigationMenu`
`.forceCommunityNavigationMenu .navigationMenuWrapper`
	`.comm-navigation`

Mobile Menu Curtain

Previous Selector	New Selector
`.forceCommunityNavigationMenu .navigationMenuWrapperCurtain`	`.comm-navigation nav.slds-is-fixed`

Home Menu Item

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

Mobile Menu Toggle Button

Previous Selector	New Selector
`.forceCommunityNavigationMenu .toggleNav`	`.siteforceServiceBody .cHeaderPanel .cAltToggleNav`

Top-Level Menu Items

Includes submenu triggers.

Previous Selector	New Selector
`.forceCommunityNavigationMenu .menuItem` `.forceCommunityGlobalNavigation .navigationMenuNode`	`.comm-navigation .comm-navigation__list > .slds-list__item`

Current Top-Level Menu Item

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

Top-Level Menu Item Links

Previous Selector	New Selector

`.forceCommunityNavigationMenu .menuItemLink`
`.forceCommunityNavigationMenu a.menuItemLink`
`.forceCommunityNavigationMenu .menuItem .menuItemLink`
`.forceCommunityNavigationMenu .menuItem a`
	`.comm-navigation .comm-navigation__list > .slds-list__item > a`
	`.comm-navigation .comm-navigation__list > .slds-list__item > button`

Previous Selector	New Selector
`.forceCommunityNavigationMenu .menuItem a.menuItemLink`	

Submenu Items

Previous Selector	New Selector
`.forceCommunityNavigationMenu .subMenuItem`	`.comm-navigation .slds-list_vertical.slds-is-nested .slds-list__item`

Current/Active Submenu Item

Previous Selector	New Selector
`.forceCommunityNavigationMenu .subMenuItem.current`	`.comm-navigation .slds-list_vertical.slds-is-nested .slds-list__item .slds-is-active`

Submenu Trigger Link

Previous Selector	New Selector
`.forceCommunityNavigationMenu .triggerLink` `.forceCommunityNavigationMenu .triggerLabel`	`.comm-navigation .slds-list__item button:enabled`

Submenu Trigger Link Icon

Previous Selector	New Selector
`.forceCommunityNavigationMenu .triggerLink .forceIcon`	`.comm-navigation .slds-list__item button:enabled .slds-icon_container`

Menu Items

Includes top-level and submenu items.

Previous Selector	New Selector
`.forceCommunityNavigationMenu .navigationMenu li`	`.comm-navigation .slds-list__item`

Menu Item Links

Includes top-level and submenu items.

Previous Selector	New Selector
`.forceCommunityNavigationMenu a` `.forceCommunityNavigationMenu a.menuItemLink`	`.comm-navigation .slds-list__item a` `.comm-navigation .slds-list__item button`

Submenus

Previous Selector	New Selector
`.forceCommunityNavigationMenu .subMenu`	`.comm-navigation .slds-list_vertical.slds-is-nested`

Submenu Items

Previous Selector	New Selector
`.forceCommunityNavigationMenu .subMenuItem`	`.comm-navigation .slds-list_vertical.slds-is-nested .slds-list__item`

Submenu Item Links

Previous Selector	New Selector
`.forceCommunityNavigationMenu .subMenuItem a` `.forceCommunityNavigationMenu .subMenu a`	`.comm-navigation .slds-list_vertical.slds-is-nested .slds-list__item a`

CSS Overrides Migration for the Panels Container

Some CSS selectors for the Panels Container changed with the Lightning communities 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.

This topic identifies selector changes for the Panels Container.

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.

Panels Container

Previous Selector	New Selector

`.uiPanelManager2`
`.onePanelManager`
`.siteforcePanelManager`
	`.comm-panels-container`

CSS Overrides Migration for the Record Banner Component

Some CSS selectors for the Record Banner component changed with the Lightning communities update. 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.

Detail Field Label

Previous Selector	New Selector
`.forceCommunityRecordHeadline` `.slds-text-heading-label-normal`	`.forceCommunityRecordHeadline` `.slds-form-element__label`

Detail Field Value

Previous Selector	New Selector
`.forceCommunityRecordHeadline` `.slds-text-body--regular`	`.forceCommunityRecordHeadline` `.slds-form-element__static`

CSS Overrides Migration for the Record Detail Component

Some CSS selectors for the Record Detail component changed with the Lightning communities update. 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.

Form Element Separator

Previous Selector	New Selector
`.forceCommunityRecordDetail .slds-form-element__control .slds-form-element_separator`	`.forceCommunityRecordDetail .slds-form-element_separator`

CSS Overrides Migration for the Record Layout Component

Some CSS selectors for the Record Layout component changed with the Lightning communities update. 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 Record Layout

Previous Selector	New Selector
`.forceRecordLayout`	`force-record-layout2`
`.forceRecordLayout.forcePageBlock`	`force-record-layout-block`

Section

Previous Selector	New Selector
`.forceRecordLayout .full.forcePageBlockSectionView`	`force-record-layout2 force-record-layout-section`
`.forceRecordLayout .full.forcePageBlockSection`	
`.forceRecordLayout .full.forcePageBlockSectionView.forcePageBlockSection`	
`.forceRecordLayout .forcePageBlockSectionView`	
`.forceRecordLayout .forcePageBlockSection`	
`.forceRecordLayout .forcePageBlockSectionView.forcePageBlockSection`	

Section Title

Previous Selector	New Selector
`.forceRecordLayout .full.forcePageBlockSection h3`	`force-record-layout2 force-record-layout-section h3`

Section Row

Previous Selector	New Selector
`.forceRecordLayout .full.forcePageBlockSectionRow` `.forceRecordLayout .forcePageBlockSectionRow`	`force-record-layout2 force-record-layout-row`

Section Item

Previous Selector	New Selector

`.forceRecordLayout .full.forcePageBlockItem.forcePageBlockItemView`
`.forceRecordLayout .full.forcePageBlockItem`
`.forceRecordLayout .full.forcePageBlockItemView`
`.forceRecordLayout .forcePageBlockItem`
`.forceRecordLayout .forcePageBlockItemView`
`.forceRecordLayout .forcePageBlockItem.forcePageBlockItemView`
	`force-record-layout2 force-record-layout-item`

Section Item Label

Previous Selector	New Selector
`.forceRecordLayout .slds-form-element__label`	`force-record-layout2 .slds-form-element__label`

Section Item Value

Previous Selector	New Selector
`.forceRecordLayout .itemBody`	`force-record-layout2 .slds-form-element__static`

Section Item Value Link

Previous Selector	New Selector
`.forceRecordLayout a`	`force-record-layout2 a`

CSS Overrides Migration for the Record List Component

Some CSS selectors for the Record List component changed with the Lightning communities update. 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.

List View Button Bar

Previous Selector	New Selector
`.forceListViewManagerHeader .forceListViewManagerButtonBar`	`.forceListViewManagerHeader force-list-view-manager-button-bar`

List View Button Bar Buttons

Previous Selector	New Selector
`.forceListViewManagerHeader .forceListViewManagerButtonBar button`	`.forceListViewManagerHeader force-list-view-manager-button-bar button`

List View Status Information

Previous Selector	New Selector
`.forceListViewManagerHeader .test-listViewStatusInfo`	`.forceListViewManagerHeader force-list-view-manager-status-info`

Picker Trigger Link

Previous Selector	New Selector
`.forceListViewManagerHeader .triggerLink`	`.forceListViewManagerHeader .triggerLink .slds-button`

CSS Overrides Migration for the Record Related List Component

Some CSS selectors for the Record Related List component changed with the Lightning communities update. 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.

Detail Field Label

Previous Selector	New Selector

`.forceCommunityRelatedRecords`
`.slds-card__header-link`
`.slds-text-heading--small`
	`.forceCommunityRelatedRecords`
	`.slds-card__header-title`

CSS Overrides Migration for the Related Articles Component

Some CSS selectors for the Related Articles component changed with the Lightning communities update. 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 Related Articles

Previous Selector	New Selector

`.selfServiceSimilarArticles`
`.base-items`
`.uiAbstractList`
`.selfServiceBaseSimpleItems`
	`.comm-related-articles`

Component Title

Previous Selector	New Selector
`.selfServiceSimilarArticles h2`	`.comm-related-articles h2`

Previous Selector	New Selector
`.selfServiceSimilarArticles .base-items-header`	

Article List

Previous Selector	New Selector
`.selfServiceSimilarArticles ul` `.selfServiceSimilarArticles .base-items`	`.comm-related-articles ul`

Article List Items

Previous Selector	New Selector
`.selfServiceSimilarArticles .base-simple-item`	`.comm-related-articles li`

Article Links

Previous Selector	New Selector
`.selfServiceSimilarArticles .item-title` `.selfServiceSimilarArticles .item-title-link`	`.comm-related-articles li a`

CSS Overrides Migration for the Reputation Leaderboard Component

Some CSS selectors for the Reputation Leaderboard component changed with the Lightning communities update. 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 Reputation Leaderboard

Previous Selector	New Selector
`.forceCommunityReputationLeaderboard.leaderboard`	`.comm-leaderboard`

Reputation Leaderboard Row

Previous Selector	New Selector
`.forceCommunityReputationLeaderboardRow`	`.comm-leaderboard li`

User Info Column

Previous Selector	New Selector
`.forceCommunityReputationLeaderboard .pointsAndLevels` `.forceCommunityReputationLeaderboard .userInfoCol`	`.comm-leaderboard .slds-media__body`

Reputation Points Column

Previous Selector	New Selector
`.forceCommunityReputationLeaderboard .reputationCol`	`.comm-leaderboard__points-column`

Title

Previous Selector	New Selector
`.forceCommunityReputationLeaderboard .title`	`.comm-leaderboard h2`

User Level Name

Previous Selector	New Selector
`.forceCommunityReputationLeaderboard .reputationLevelName`	`.comm-leaderboard__level-name`

User Reputation Level Image

Previous Selector	New Selector
`.forceCommunityReputationLeaderboard .reputationLevelImage`	`.comm-leaderboard .slds-media__body .slds-icon_small`

Username in User

Previous Selector	New Selector
`.forceCommunityReputationLeaderboard .userName`	`.comm-leaderboard__user-name`

User Photo in User

Previous Selector	New Selector
`.forceCommunityReputationLeaderboard .userPhoto`	`.comm-leaderboard .slds-media__figure`

User Points in Reputation Levels

Previous Selector	New Selector
`.forceCommunityReputationLeaderboard reputationPointsNumber.`	`.comm-leaderboard__points`

User Points Word in Reputation Levels

Previous Selector	New Selector
`.forceCommunityReputationLeaderboard .reputationPointsWord.`	`.comm-leaderboard__points-word`

CSS Overrides Migration for Trending Articles by Topic Component

Some CSS selectors for the Trending Articles by Topic component changed with the Lightning communities update. 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 Trending Articles

Previous Selector	New Selector
`.selfServiceTopicTrendingArticles .base-items`	`.comm-topic-trending-articles`

Previous Selector	New Selector
`.uiAbstractList` `.selfServiceBaseSimpleItems`	

Component Title

Previous Selector	New Selector
`.selfServiceTopicTrendingArticles h2` `.selfServiceTopicTrendingArticles .base-items-header`	`.comm-topic-trending-articles h2`

Article List

Previous Selector	New Selector
`.selfServiceTopicTrendingArticles ul` `.selfServiceTopicTrendingArticles .base-items`	`.comm-topic-trending-articles ul`

Article List Items

Previous Selector	New Selector
`.selfServiceTopicTrendingArticles .base-simple-item`	`.comm-topic-trending-articles li`

Article Links

Previous Selector	New Selector
`.selfServiceTopicTrendingArticles .item-title-link` `.selfServiceTopicTrendingArticles a`	`.comm-topic-trending-articles li a`

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 then select **Static Resources**.

2. Click **New**, upload the file, and give the static resource a name. Keep a note of the resource name.

   If your community has public pages, select **Public** in the Cache Control setting. If you don't make the font resource publicly available, the page uses the browser's default font instead.

3. In Community Builder, open the CSS editor by clicking **Theme** > > **Edit CSS**.

4. Use the `@font-face` CSS rule to reference the uploaded font.
   For example:

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

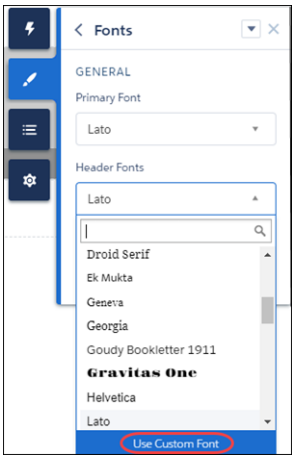
   - To reference a single font file, use the syntax *`path_prefix`*`/s/sfsites/c/resource/`*`resource_name`*, where *`path_prefix`* is the URL value added when you created the community, such as `myPartnerCommunity`.

     For example, if you upload a file called `myFirstFont.woff` and name the resource MyFonts, the URL is `myPartnerCommunity/s/sfsites/c/resource/MyFonts`.

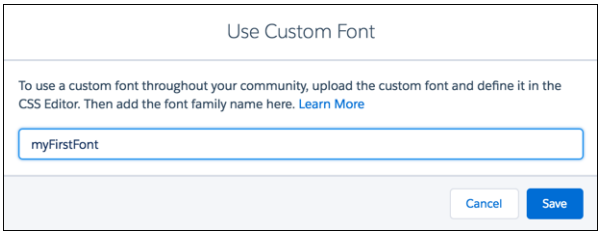
   - To reference a file in a .zip file, include the folder structure but omit the .zip file name. Use the syntax *`path_prefix`*`/s/sfsites/c/resource/`*`resource_name`*`/`*`font_folder`*`/`*`font_file`*.

     So if you upload `fonts.zip`, which contains `bold/myFirstFont.woff`, and you name the resource MyFonts, the URL is `myPartnerCommunity/s/sfsites/c/resource/MyFonts/bold/myFirstFont.woff`.

5. In the Theme panel, select **Fonts**, select the Primary Font or Header Fonts dropdown list, and then click **Use Custom Font**.



6. Add the font family name that you entered in the CSS editor—for example, `myFirstFont`—and save your changes.



Customize the Theme Layout of Your Template

To put your own stamp on a template theme 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 component is the top-level layout for the template pages (1) in your community. Theme layout components are organized and applied to your pages through theme layouts. Theme layout components include the common header and footer (2), and often include 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?

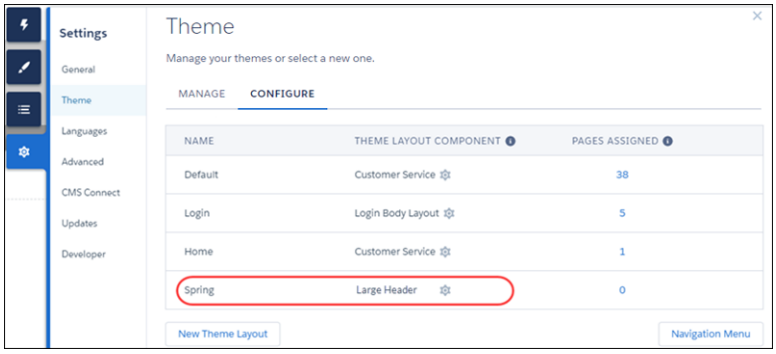
To understand how a theme layout works, let's look at things from the Community Builder perspective. In Community Builder, theme layouts combine with theme layout components to give you granular control of the appearance and structure of each page in your community. You can customize the layout's header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

A theme layout categorizes the pages in your community that share the same theme layout component. You can assign a theme layout component to any existing theme layout. Then you apply the theme layout —and thereby the theme layout component—in the page's properties.

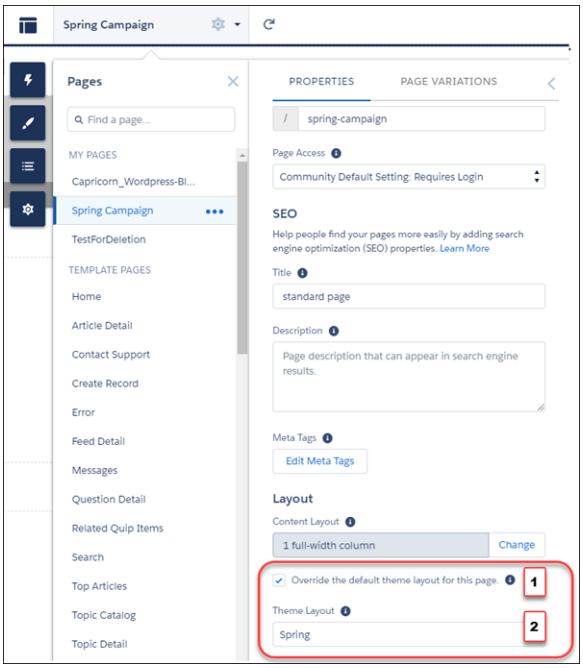
Customer Service includes the following theme layouts and components, but you can create custom components or switch layouts as needed.

- Login applies the theme layout component, Login Body Layout theme layout to the login pages.
- Default is applied to all non-login related pages.

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 called Spring to categorize the campaign pages and you assign the Large Header layout component to it.



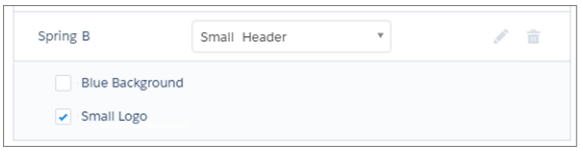
Next, you apply the Spring theme layout in each page's properties, which instantly applies the Large Header layout to each page. Select **Override the default theme layout for this page.** (1) to display Theme Layout. Choose the new layout (2) from the available choices.



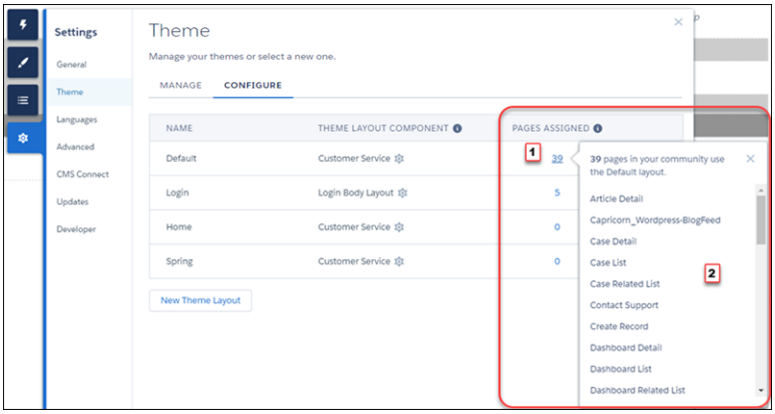
Example: Everything looks rosy until the VP of marketing decides that the header takes up too much room. That's an easy fix, because you don't have to update the properties of each page to change the theme layout. Instead, with one click in the Theme area, you can switch Spring to the Small Header layout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

Example: Now let's say that the Small Header layout includes two custom properties—Blue Background and Small Logo—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 could create a theme layout called Spring B, assign the Small Header layout component to it, and enable Small Logo. Then, you apply the Spring B theme to the page.



Not sure which pages are associated with any of your Theme Layouts?



With a glance and a click, you can see how many and which pages are associated with any of your theme layouts. From **Settings** > **Theme**, click the Pages Assigned total shown for any theme layout row (1) to open a list of the pages associated with that theme layout (2) before making changes.

Theme layouts make it easy to reuse the same theme layout component 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 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.

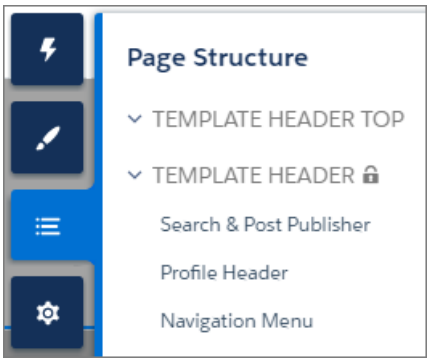
Add attributes declared as `Aura.Component[]` to include regions in the theme layout, which contain the page's components. You can add components to the regions in your markup or leave regions open for users to drag-and-drop components into. Attributes

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, 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

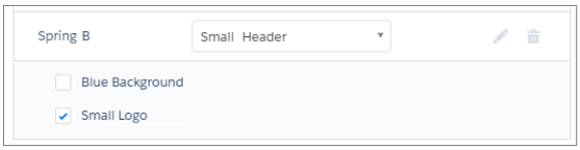
You can expose theme layout properties in Community Builder by adding a design resource to your bundle.

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`.

SEE ALSO:

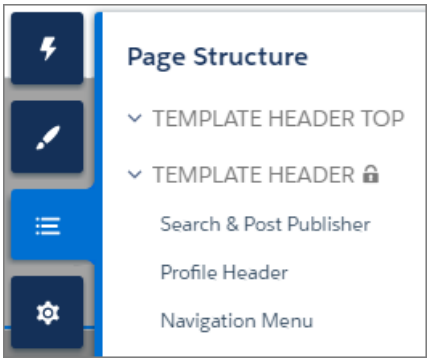
Salesforce Help: Change the Content Layout in Community Builder

Configure Swappable Search and Profile Menu Components

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

In Customer Service, 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 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 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>
```

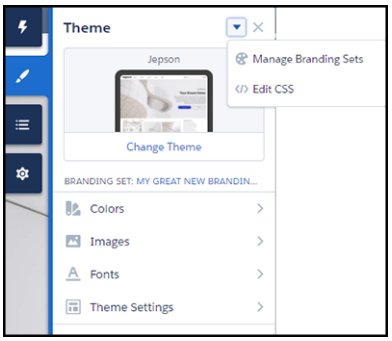
SEE ALSO:

Step 5: Build a Custom Search Component

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 Theme panel in Community Builder.

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



Available Tokens for Communities

For Communities using the Customer Service 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	`colorTextDefault`
Detail Text Color	• `colorTextLabel`

These Branding panel properties...	...map to these standard design tokens
	• `colorTextPlaceholder` • `colorTextWeak`
Action Color	• `colorBackgroundButtonBrand` • `colorBorderBrand` • `colorBorderButtonBrand` • `colorBrand` • `colorTextBrand` Note: As of Summer '18 `colorBackgroundHighlight` is no longer mapped to Action Color.
Link Color	`colorTextLink`
Overlay Text Color	• `colorTextButtonBrand` • `colorTextButtonBrandHover` • `colorTextInverse`
Border Color	• `colorBorder` • `colorBorderButtonDefault` • `colorBorderInput` • `colorBorderSeparatorAlt`
Primary Font	`fontFamily`
Text Case	`textTransform`

In addition, the following standard tokens are available for derived theme properties in the Customer Service template. You can indirectly access derived branding properties when you update the properties in the Theme panel. For example, if you change the Action Color property in the Theme 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)	• `colorBackgroundButtonBrandActive` • `colorBackgroundButtonBrandHover`
Hover Color (Derived from Action Color)	• `colorBackgroundButtonDefaultHover` • `colorBackgroundRowHover` • `colorBackgroundRowSelected` • `colorBackgroundShade`
Link Color Darker (Derived from Link Color)	• `colorTextLinkActive` • `colorTextLinkHover`

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

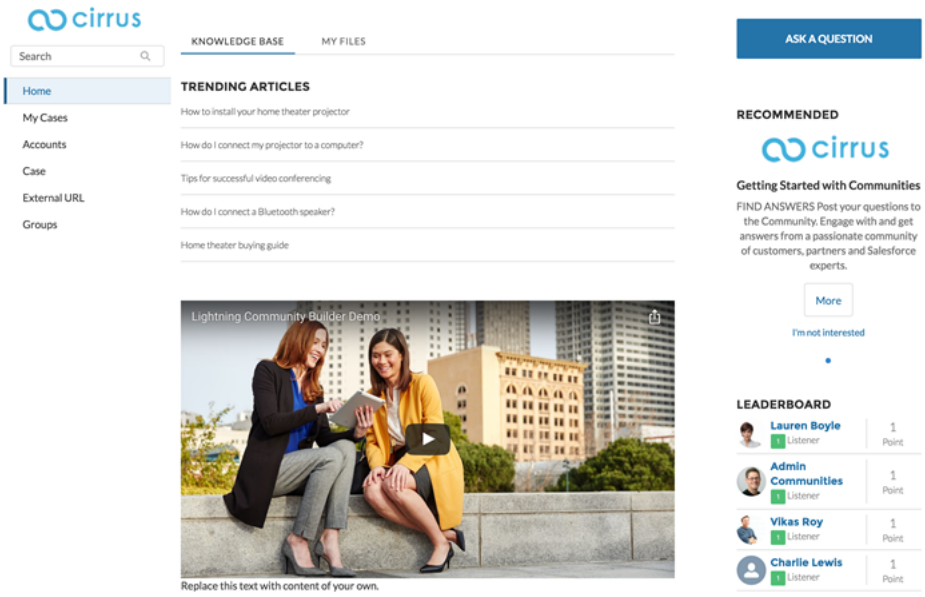
Lightning Component Developer Guide: Using the Salesforce Lightning Design System in Apps

CHAPTER 4 Example: Build a Condensed Theme Layout Component

In this chapter ...

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 component, 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>
```

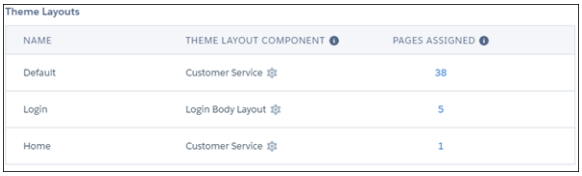
Right now, all the regions flow vertically, so add some semantic structure using the SLDS grid system.

```
<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>
```

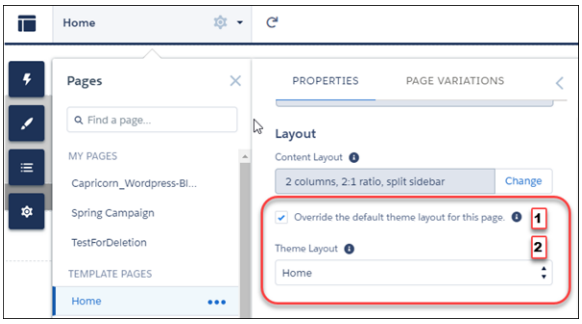
Add a design resource to the bundle to give the component a UI label.

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

</design:component>
```

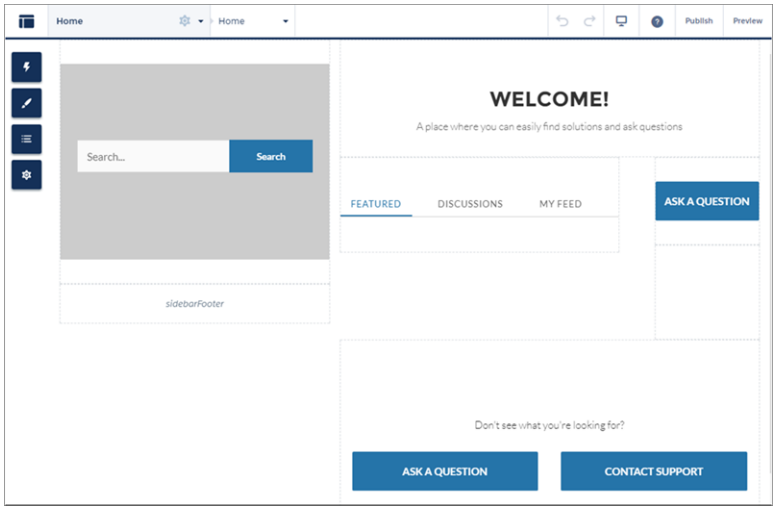
In Community Builder, you can see the theme layout's semantic hierarchy by selecting it for the Home theme layout.



Open the Page Properties for the Home page. Select **Override the default theme layout for this page.** (1) to display Theme Layout. Select **Home** in the Theme Layout dropdown (2).



The page refreshes, and now you can see the new theme layout component 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="forceCommunity:base"` in your markup, you now have access to all the tokens exposed by SLDS and the branding values defined in the Theme 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.

- Referencing the design token as part of the background image property through your CSS
- Creating a custom component that displays a static resource—for example, `{!$Resource.MyJavascriptFile}`
- Create a custom component that fetches the path to the asset from the server
- Use an out-of-the-box component, such as the Rich Content Editor

If you want to use the same company logo image that was uploaded in the Theme panel, the easiest and most consistent way is to use the design token. Let's look at that code first.

In the following CSS snippet from a component's style bundle, the code uses the `brandLogoImage` token (wrapped by a t()) to inject the company logo in the CSS as the background image of the logoContainer.

```
.THIS .logoContainer {
  background-image: t('url(' + brandLogoImage + ')');
  background-position: center;
  background-repeat: no-repeat;
  background-size: contain;
  width: 80%;
  height: 50px;
}
```

For this next 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">
    <img src="{!$Resource.cirruslogo}"/>
</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 `<li>` 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;
}
```

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>
</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 sobj : sobjs) {
        ids.add(sobj.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();
```

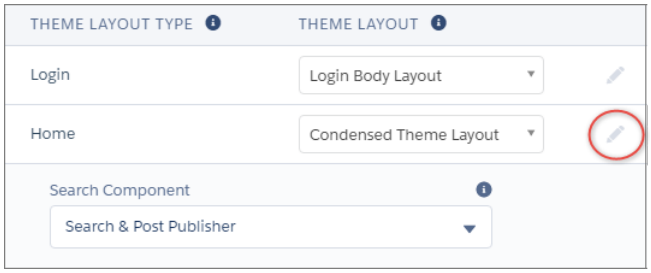
In the CSS for the component, add the following CSS rules.

```
.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" />
```

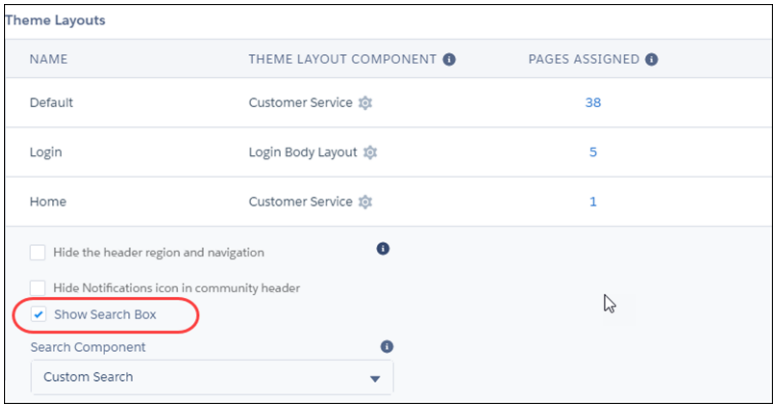
In the markup, wrap the entire search column with an `aura:if` expression. This expression is reactive, so when the attribute gets updated, the component rerenders.

```
<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.



CHAPTER 5 Develop Secure Code: Locker Service and Stricter CSP

In this chapter ...

When you develop custom Lightning components or add head markup to your community, you need to be aware of Locker Service and the stricter Content Security Policy (CSP) critical update. The Locker Service 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.

Locker Service 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.

Locker Service is enforced the same way across all orgs. However, stricter CSP uses a separate critical update for Communities, which is documented more thoroughly here.

Locker Service in Communities

Locker Service is a powerful security architecture for Lightning components that enhances security by isolating Lightning components in their own namespace. Locker Service 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.

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

You can disable Locker Service 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 Locker Service when developing Lightning components, see "What is Locker Service?" in the *Lightning Component Developer Guide*.

For information on preparing your Lightning components code for Locker Service enablement, see "Salesforce Lightning CLI (Deprecated)."

SEE ALSO:

Lightning Component Developer Guide: Browser Support Considerations for Lightning Components

Content Security Policy 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.

Note: In Winter '19, the "Enable Stricter Content Security Policy for Lightning Components in Communities" critical update was replaced with new CSP options in **Settings** > **Security** in Community Builder.

There are three levels of script security, providing enough flexibility to prevent affecting existing communities and code.

Script Security Level	Description
Strict CSP: Block Inline Scripts and Script Access to All Third-party Hosts	Recommended—provides maximum security. • Blocks all inline scripts from running in your site • Allows non-script resources, such as images, from approved third-party hosts to display
Allow Inline Scripts and Script Access to Whitelisted Third-party Hosts	Provides moderate security. • Blocks script hosts that are not explicitly whitelisted • Allows non-script resources, such as images, from approved third-party hosts to display
Allow Inline Scripts and Script Access to Any Third-party Host	Default—provides no added security but enables your community to work as currently designed. • Blocks nothing • Allows access to all third-party hosts without the need for whitelisting Note: In Winter '20 (October '19), this option is being removed.

Stricte 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.

What Do I Need to Do and When?

To ensure that your community works as expected, the default CSP setting (Allow Inline Scripts and Script Access to Any Third-party Host) maintains the same security level as before Winter '19.

However, in Spring '19 (February 2019), Strict CSP becomes the standard CSP setting for all *new* communities. And in Winter '20 (October 2019), the current default setting is being removed for existing communities. Therefore, you must plan ahead, consider the level of security your community requires, and decide when to take the required steps.

Note: We recommend using Chrome to develop your site when possible because we added extra error guidance when viewing your site. After you set the CSP security level for your community, test it in other browsers to ensure a good experience for your customers.

SEE ALSO:

Lightning Component Developer Guide: Content Security Policy Overview

Lightning Component Developer Guide: Stricter CSP Restrictions

Salesforce Help: Script Level Security in Communities

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

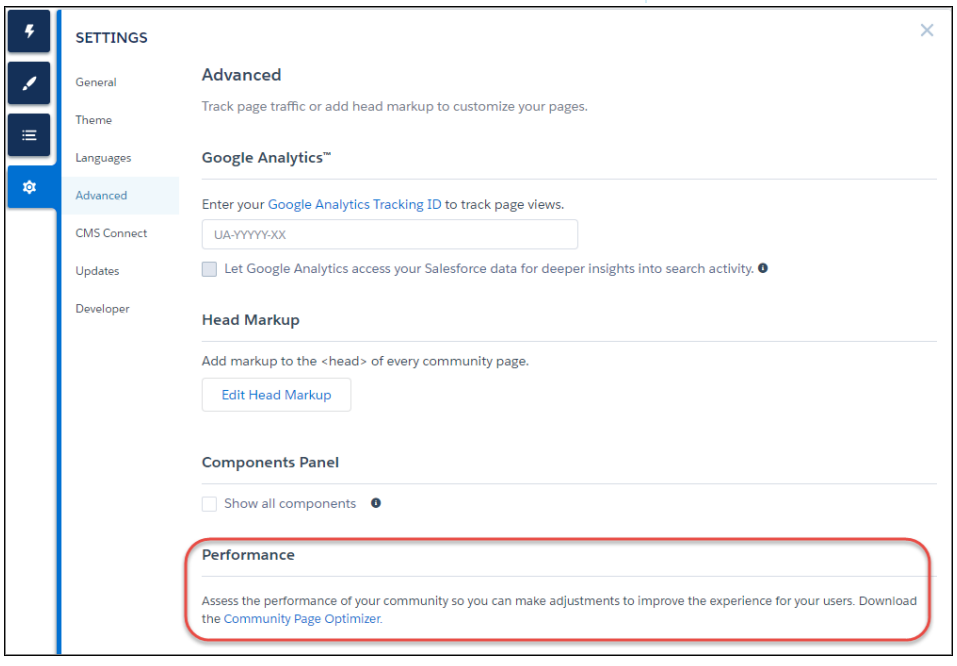
Available in: Salesforce Classic (not available in all orgs) and Lightning Experience

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

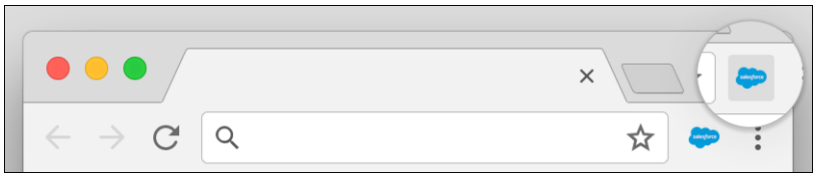
USER PERMISSIONS

To create, customize, or publish a community:

- Create and Set Up Communities AND View Setup and Configuration

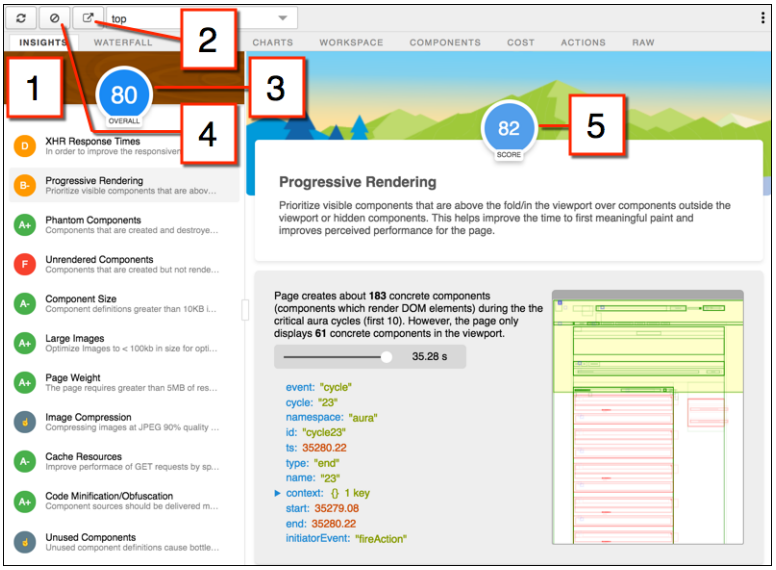


After installation, the Community Page Optimizer is located with your other Chrome extensions.



Insights

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



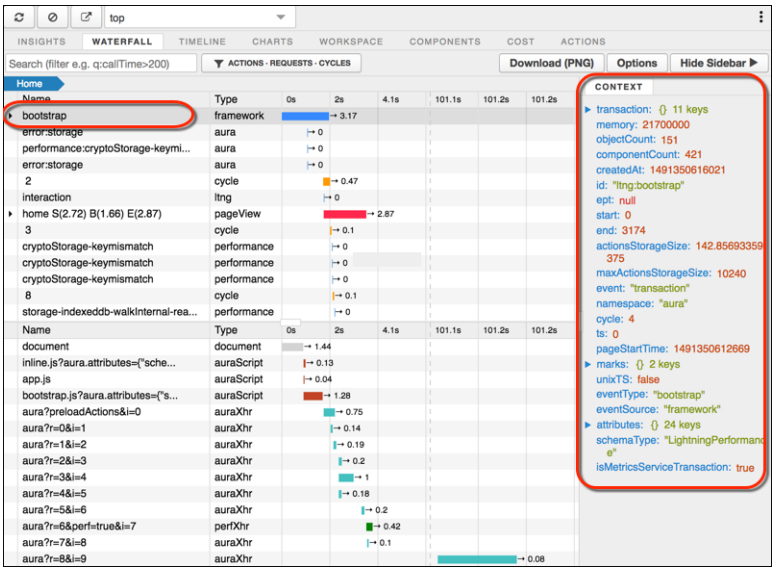
The Insights tab (1) evaluates your page based on best practices for web applications developed using the Lightning framework. This tab displays an overall performance score (3) along with individual scores (5) for various analysis rules. To view details and suggested actions, click each rule. Click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

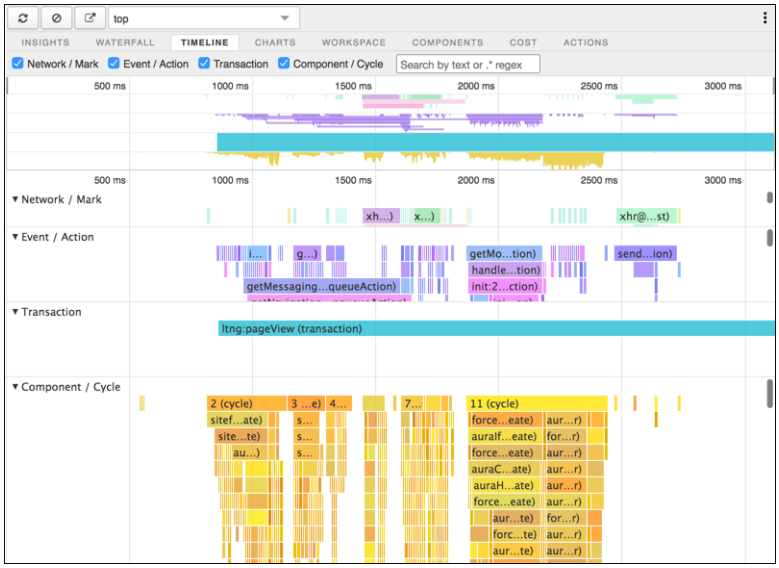
Waterfall

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.



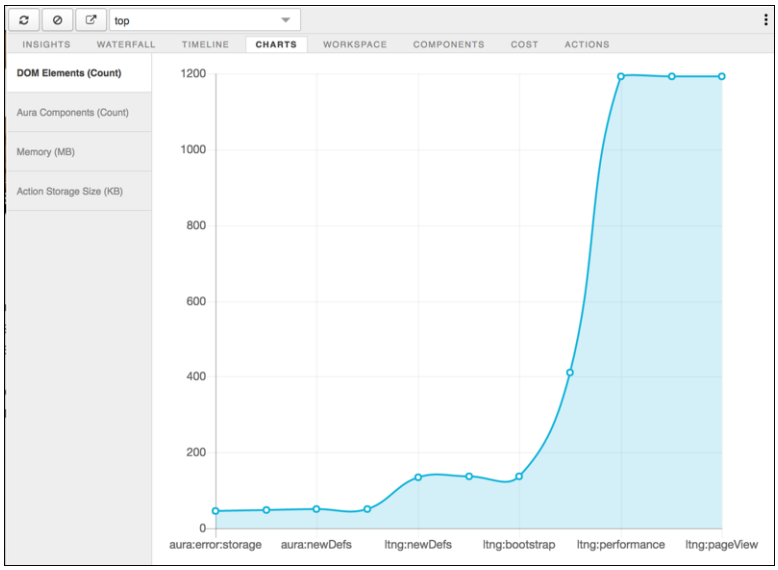
Timeline

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.



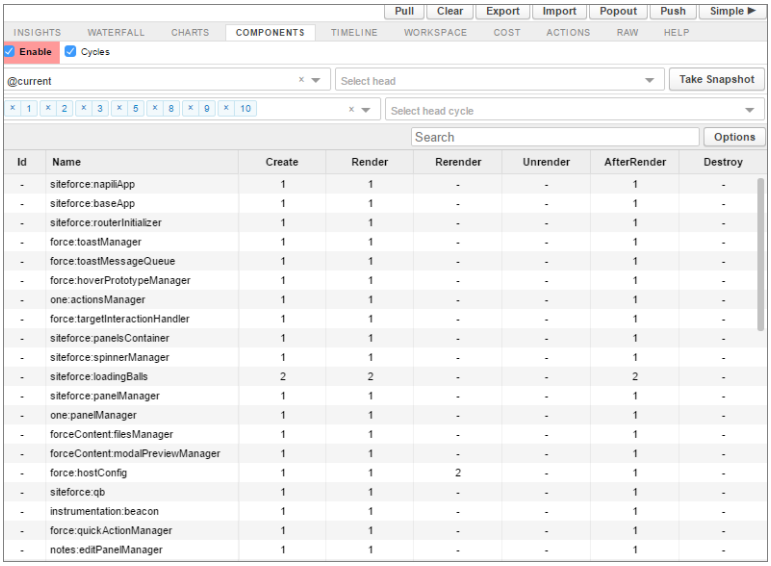
Charts

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



Components

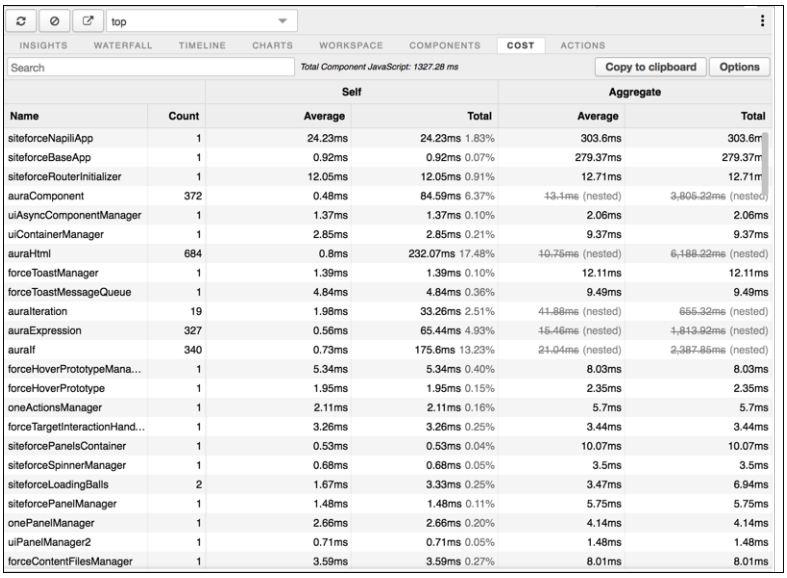
The Components tab displays the life cycle counts for each component on the page. This view helps you identify potential component leaks and unexpected rendering behavior. Use the Component tab along with the Cost tab for an overall view of component performance.



Id	Name	Create	Render	Rerender	Unrender	AfterRender	Destroy
-	siteforce:napiliApp	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.

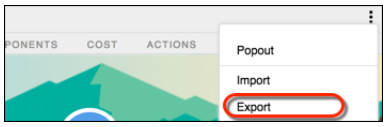


Actions

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 ...

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, CSS, and JavaScript to customize your community and keep its branding and other content consistent with your website.

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 Adobe Experience Manager, 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. You have many configuration options, including language mapping between your CMS and Salesforce, determining the load order of multiple connections, and specifying the CSS scope.

CMS Connect is available in communities that are based on Customer Service, 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:

- `<img>` 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
- `<del>`, `<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 **Content Management**.
3. Click **CMS Connect**.
4. Click **New CMS Connection** (if no connections have been created yet in your community) or **New**.
5. 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.)
6. Select your CMS source.

   Note: If your CMS server isn't listed, select Other. CMS Connect works with the HTML, JSON, CSS, and HTTP standards and isn't provider-specific.

7. For Server URL, enter the full path to a CMS server that's accessible using HTTPS (HTTP over SSL). Use a fully qualified domain name, such as: `https://www.example.com`.
8. 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, here's an example of a root path to content in AEM.

   ```
   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.

9. If your CMS source is Adobe Experience Manager (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.

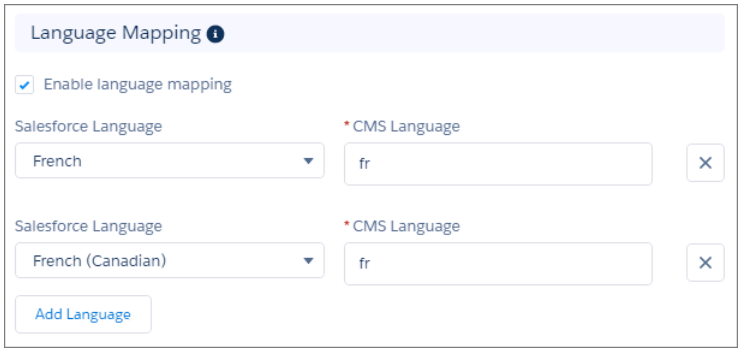
   > Note: To use personalization, enable it for the components you want to personalize. Do that in Builder Settings for header and footer, and in Community Builder for banners or other components.

10. 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. Style sheets load in the order listed. Use the up and down arrows to change the order.
11. 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.
12. 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 a Connection for Your JSON CMS for details.
13. 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 the language in which the CMS content is displayed. For example, if your community has French and French Canadian enabled, you can set it up so the French Canadian community displays French content:



14. 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.

Reuse Content with CMS Connect JSON

Want to reuse blogs and articles that you've already published in WordPress, Drupal, or some other CMS? Easy! You can bring in JSON content to your community pages using CMS Connect.

Set Up a Connection for Your JSON CMS

To set up a CMS connection to your JSON content, specify the server URL and paths to the content. The CMS connection is the secret sauce that lets your community page locate and retrieve CMS content.

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.

Reuse Content with CMS Connect JSON

Want to reuse blogs and articles that you've already published in WordPress, Drupal, or some other CMS? Easy! You can bring in JSON content to your community pages using CMS Connect.

Content Item or Content List?

Suppose that you want to reuse JSON content that's in your CMS and display it in your community. CMS Connect supports two basic types of JSON content. What they're called in one CMS can differ from what another CMS calls them. To keep things simple, in CMS Connect, 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 a series of blogs. When a content list displays on a page, you see a title and summary for each list item, with a link to the full blog.

How to Reuse JSON Content

The process boils down to three key steps.

1. If needed, set up Cross-Origin Resource Sharing (CORS). If your CMS resources aren't available to your community, add your community host to the trusted hosts in the CORS header of your CMS system. All CMS connections must use unauthenticated HTTPS (HTTP over SSL).
2. Create a CMS connection. When you set up the connection, you define whether the content is a content item or a content list and specify the server URL and content paths.
3. Add CMS Connect (JSON) components to your community pages on page 72. Using the Community Builder, drag the component to your community page and configure its properties. Use JSON expressions to bring in the title, author, blog content, and so on from your CMS.

Preview Your JSON Content

It helps to preview the JSON response body so that you understand the data format of your CMS system. For example, suppose that the URL for your WordPress website is `https://capricornblog.wordpress.com`. Enter the full JSON endpoint for the CMS content in a browser, for example,
`https://public-api.wordpress.com/rest/v1.1/sites/capricornblog.wordpress.com/posts`.

You use this endpoint when you define the CMS connection. The endpoint has two parts: the server URL and a JSON path. In this example, the server URL points to a WordPress site, and the JSON path is the path for a set of blogs. When you configure a CMS Connect (JSON) component, you retrieve the blog series as a content list, and each individual blog as a content item.

When you enter the endpoint in a browser, you see the data format of the JSON response body. Previewing the response helps you understand how to configure the CMS Connect component on your community pages.

Example Drupal Responses

Example: Let's say you have JSON content in a Drupal CMS. Here's what the JSON response body for an example 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"
    }
   }
  }
```

```
  }
}
```

The path to the content item is structured as `{baseUrl}/jsonApi/node/blogs/{id}`. The JSON object *`{baseUrl}`* is a variable containing the server URL for the content. The path to the content is `jsonApi/node/blogs/{id}`. The elements in braces ({ }) act as variables that pass values at runtime.

In this example, `@data` points to the parent node. Attribute nodes for `@data` are nested below the parent.

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

When you configure a CMS Connect (JSON) component, use JSON expressions to specify the connection and content that you want to retrieve.

For example, this JSON expression pulls the title of the content.

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

This expression retrieves the content ID.

```
@data/id
```

And this JSON expression retrieves the content body, for example, the blog text.

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

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 `{baseUrl}/jsonApi/node/page`.

```
{
  "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"
```

```
    }
}
```

You can display JSON content from multiple sources on your community pages. Set up JSON connections in your CMS Connect workspace. Then drag components to your pages and define JSON expressions that pull in your CMS content.

SEE ALSO:

Set Up a Connection for Your JSON CMS
Add CMS Connect (JSON) Components to Your Community Pages
Example: Connect JSON Content to Your Community

Set Up a Connection for Your JSON CMS

To set up a CMS connection to your JSON content, specify the server URL and paths to the content. The CMS connection is the secret sauce that lets your community page locate and retrieve CMS content.

1. In Setup, enter `All Communities` in the Quick Find box, then select **All Communities**.
2. Under Workspaces, click **Content Management** and then select **CMS Connect**.
3. To create a connection, click **New**, and enter the connection details.
   - **Name**—Enter a connection name, for example, `Blogs`.
   - **Connection Type**—Select the connection type. To use an authenticated connection, first define a named credential for your CMS site.
   - **CMS Source**—Select your CMS source, for example, **WordPress**.
   - **Server URL**—Enter the URL for a CMS server that's accessible using HTTPS. Use a fully qualified domain name, for example, `https://public-api.wordpress.com`.

   You can also select an existing connection to edit and add JSON content.

4. In the JSON section, click **Add JSON**.

   Note: When defining a JSON connection, skip the Root Path, CSS, and Scripts sections. They're needed only for HTML connections.

5. Enter a content type name. It can be anything you want. For example, `Home Improvement Blogs`. The name provides an easy way to group related content.
6. Select the type of content to add. Is it a single article or blog post, such as DIY Dryer Vent Cleaning? Then select **Content Item**. Or is it a grouping of items, such as DIY Featured Blogs, with links to individual blog posts? Then select **Content List**.

   Let's create a content list that pulls in some blog articles. When you configure a content list, you also set up a content item that points to each item in the list.

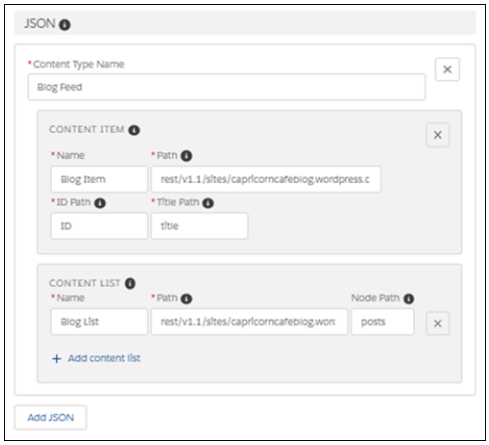
7. Click **Add a content list**.
   a. Enter a name for the list, for example, `Featured Blogs`.
   b. Enter the relative path to the JSON content in your CMS, for example, `rest/v1.1/sites/capricornblog.wordpress.com/posts`.

   c. For Node Path, enter a JSON expression for the starting node in the list. For example, `posts`. Because a JSON expression is expected here, you don't enter an @.

8. Click **Add a content item**.

> Note: To create a detail page and a Read More link for your list items, add a content item with your content list under the same content type name. When you publish your community changes and click on the link for the detail page, it displays the corresponding item in full. For example, clicking the link under an item in a blog series displays the full blog article from your CMS, not just a summary.

   a. Enter a name for the content item, for example, `Blog Item`.
   b. Enter the relative path, for example, `rest/v1.1/sites/capricornblog.wordpress.com/posts/{component}`. The `{component}` variable indicates the path for an item ID, like `$postId`. It is appended dynamically at run time based on the content item referenced. There are four variables you can use in a JSON path: `{component}`, `{itemsPerPage}`, `{pageNumber}`, and `{offset}`.
   c. Enter the ID and title paths for the content item. JSON expressions are expected in these paths, so don't enter them with an @.



9. Want to add more JSON served content? Repeat the previous steps for each content item or content list to add. You can add up to five different JSON connections, each with up to one JSON content item and 10 JSON content lists. Save the connection when you're finished.
10. If your content has multiple languages, select **Enable language mapping**, and add at least one language. When you specify a path to a content item or list, you can use the `{language}` variable to point to content in another language.

The next step is to drag a JSON component to your page using the Community Builder. For instructions, see Add CMS Connect (JSON) Components to Your Community Pages.

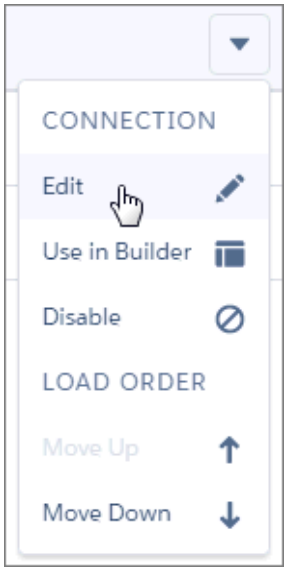
SEE ALSO:

- Reuse Content with CMS Connect JSON
- Add CMS Connect (JSON) Components to Your Community Pages
- Example: Connect JSON Content to Your Community

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 on the line of 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
Ready to show external JSON articles and blogs on your community pages? To display your content just the way you want, use the property editor in the Community Builder to customize your component layout.

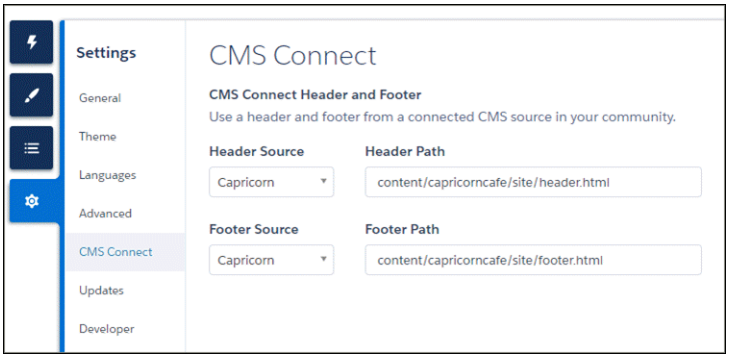
CMS Connect (JSON) Expressions
Reusing your content is a great way to save time and effort. By defining JSON expressions for a CMS Connect (JSON) component, you can map existing content into your community pages.

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.

Before you can add CMS headers and footers, set up your CMS connection on page 65.

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 65.

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

Add CMS Connect (JSON) Components to Your Community Pages

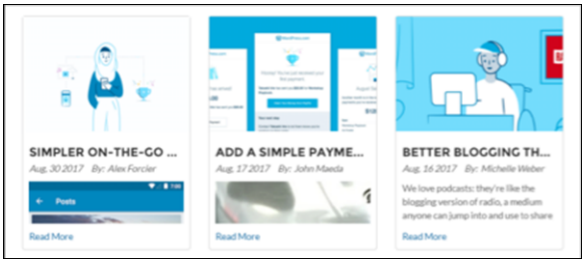
Ready to show external JSON articles and blogs on your community pages? To display your content just the way you want, use the property editor in the Community Builder to customize your component layout.

Before you can add JSON components to your community pages, set up connections to your JSON CMS content on page 72. The connection defines the content lists and content items to display. It also specifies the path to your content.

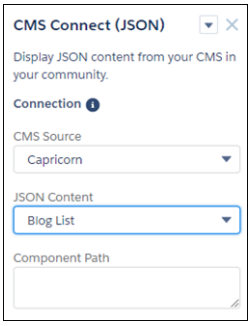
After you set up the connection, you're ready to configure how the content appears on your community pages.

1. In Setup, enter `All Communities` in the Quick Find box, then select **All Communities**.
2. Next to the community name where you want to add CMS content, click **Builder**.
3. Click to open the Component panel.
4. Drag a CMS Connect (JSON) component to where you want to display it on the page.
5. In the component's property editor, configure component properties.

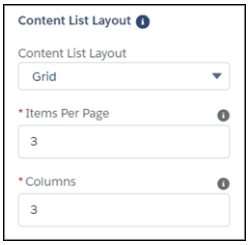
Example: Let's say you want to display a list of blogs on your page.



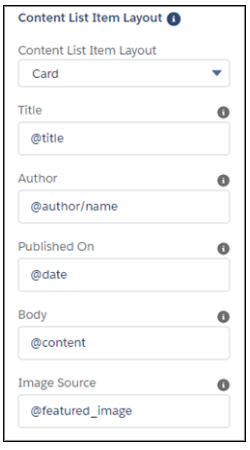
1. In the property editor, select the CMS source you defined as a connection. Select a content list as your JSON content.



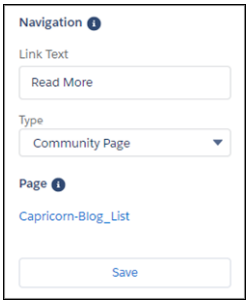
2. Define the grid layout for your list.



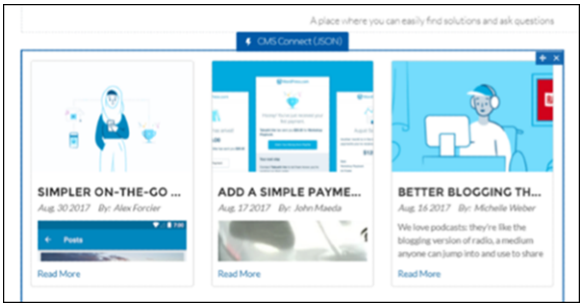
3. To configure the layout of each list item, enter JSON expressions in that section.



4. Configure how you display each blog item, either through a community page or an external link.



After you save the settings, you see a preview of the content in the page area.



Detail Page

Notice the Read More link for each item? After you publish your changes, you can click a link under a blog item in the series. When you click the link, you see an external page or a community page that's created dynamically for each content item in your content list. To change the name or URL of this detail page, locate the page in the Pages menu in the top toolbar, and modify its properties. For search engine optimization (SEO), detail page URLs are added to the sitemap files.

For each content type name, only one detail page is created. If you create more than one content list under the same content type name, you see the same detail page.

SEE ALSO:

- Reuse Content with CMS Connect JSON
- Set Up a Connection for Your JSON CMS
- CMS Connect (JSON) Expressions
- Example: Connect JSON Content to Your Community

CMS Connect (JSON) Expressions

Reusing your content is a great way to save time and effort. By defining JSON expressions for a CMS Connect (JSON) component, you can map existing content into your community pages.

When you place a component on your community page, you specify its properties using JSON expressions. For example, if you are mapping in a blog article, you can define expressions to retrieve the blog's ID, title, author, and content from your CMS.

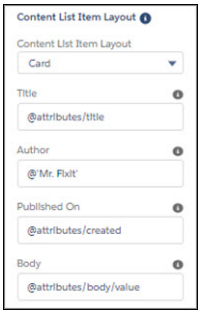
Let's look at an example using this JSON structure as the content source. The example JSON response body is from a Drupal CMS.

```
{
 "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": "DIY Dryer Vent Cleaning",
   "created": 1502133909,
   "changed": 1502133933,
   "body": {
    "value": "This is the body of the blog article. Step 1...",
    "summary ": "How to clean your dryer vent in 3 easy steps."
   }
  },
  "relationships": {
   "type ": {
    "data ": {
     "type": "node_type--node_type",
     "id": "5b80bc9e-dc78-4612-add8-e46b2e2ff616"
    }
   }
  }
 }
}
```

To set up a CMS connection, create a content item (for example, a blog) or a content list (a blog series). When you add the component to your page, you enter JSON expressions to define its properties.

For example, to retrieve the title, the expression `@data/attributes/title` retrieves DIY Dryer Vent Cleaning from the example JSON resource. The @ symbol indicates that the text represents a JSON expression that points to the title element.



Basic JSON Expressions

For CMS JSON connections, the expression syntax follows the JSON Pointer specification in RFC 6901. For the example content source, the parent node `@data` is the top-level pointer to the source content. To access a nested element, just specify the path. For example, the expression `@data/attributes/body/value` points to the blog content.

When accessing values in an array, the starting index value is 0. Here's an example of JSON source containing a simple array.

```
{
 "array": [
    {
     "key": 1,
```

```
     "dictionary": {
        "a": "Air gap",
        "b": "Belt",
        "c": "Clog",
        "d": "Drain"
  },
      "var": 2
    },
    {
      "key": 2,
      "dictionary": {
        "a": "Appliance",
        "b": "Breaker",
        "c": "Coffeemaker",
        "d": "Dryer"
      },
      "var": 3
    }
  ]
}
```

Using this source, the expression on the left yields the output on the right.

Example Expression	Output
`@/array/0/key`	`1`
`@/array/1/key`	`2`
`@/array/1/dictionary/a`	`"Appliance"`

Using Functions in JSON Expressions

You can build more complex expressions using the `where`, `lookup`, and `concat` functions. Use the `where` function to filter array elements. The function compares two values using an operator: `'eq'` (equals), `'ne'` (not equals), `'lt'` (less than), `'gt'` (greater than), `'le'` (less than or equals), or `'ge'` (greater than or equals).

Example Expression	Output
`@/array[where(key,'eq',2)]/0/dictionary/b`	`"Breaker"`
`@/array[where(key,'eq',2)][where(var,'eq',3)]/0/dictionary/d`	`"Dryer"`

In the first example, the `where` expression filters the array to find where the value of key is equal to 2. The second example uses two `where` functions, and both conditions must be met to find the array element of interest.

The `lookup` function is useful when searching a JSON document. It takes at least three parameters:

- Starting node
- Key to examine inside the starting node
- String or numeric value to match the key against

For example, you can search the JSON source for instances where an author ID or a key matches a certain value.

Example Expression	Output
`@lookup(/array,key,2)/0/var`	`3`
`@lookup(/array,key,1)/0/dictionary/a`	`"Air gap"`
`@lookup(/array,key,2)/0/dictionary/a`	`"Appliance"`

The `concat` function concatenates the strings and parameters you give it. The expression `@concat('DIY Focus', ': ', 'about your ' , /array/1/dictionary/d)` outputs `"DIY Focus: about your Dryer"`. If your CMS stores the author name in an array with first and last name elements, to output the author's full name, use an expression such as `@concat(author/first_name,' ',author/last_name)`.

The Fine Print

Keep these points in mind when you compose JSON expressions.

- Enclose strings in single quotes, for example, @'hello' is how you include the string `hello` in an expression. To include an @ character in a string, enclose it in single quotes, for example, `'myname@gmail.com'`.
- Use `true` or `false` to indicate a Boolean value. The value `true` is a Boolean, but `'true'` is a string.
- Specify a sibling or parent element relative to the current context using a period (.) for a sibling and two periods (..) for a parent. For example, to search by title, specify `@data/attributes/title/../id` to retrieve the corresponding ID for a title.
- Use the backslash character (\) as an escape character. For example, `@/author/name` looks for name under an author element, but `@/author\/name` looks for author/name as a key.

Error Scenarios

Mistakes happen sometimes. Here are a couple of possible error messages when your JSON expressions aren't right.

- Entering an invalid JSON path in the property editor of the Community Builder. For example, if you enter a JSON expression for author as `@author/invalid` instead of `@author/name`, you see an error message like this one.

- Entering an expression that returns more than one value when only one value is expected. For example, if you enter a JSON expression for author as `@author` instead of `@author/name`, you see this error message.

- Providing an invalid URL path when you create a CMS connection for a content list or connection details. Depending on the issue, you see HTTP errors (for example, HTTP 0 or HTTP 404 errors) or an error like this one. Verify the configuration settings for your CMS connection.

SEE ALSO:

- Example: Connect JSON Content to Your Community
- Set Up a Connection for Your JSON CMS
- Add CMS Connect (JSON) Components to Your Community Pages
- *Metadata API Reference:* CMSConnectSource API

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.

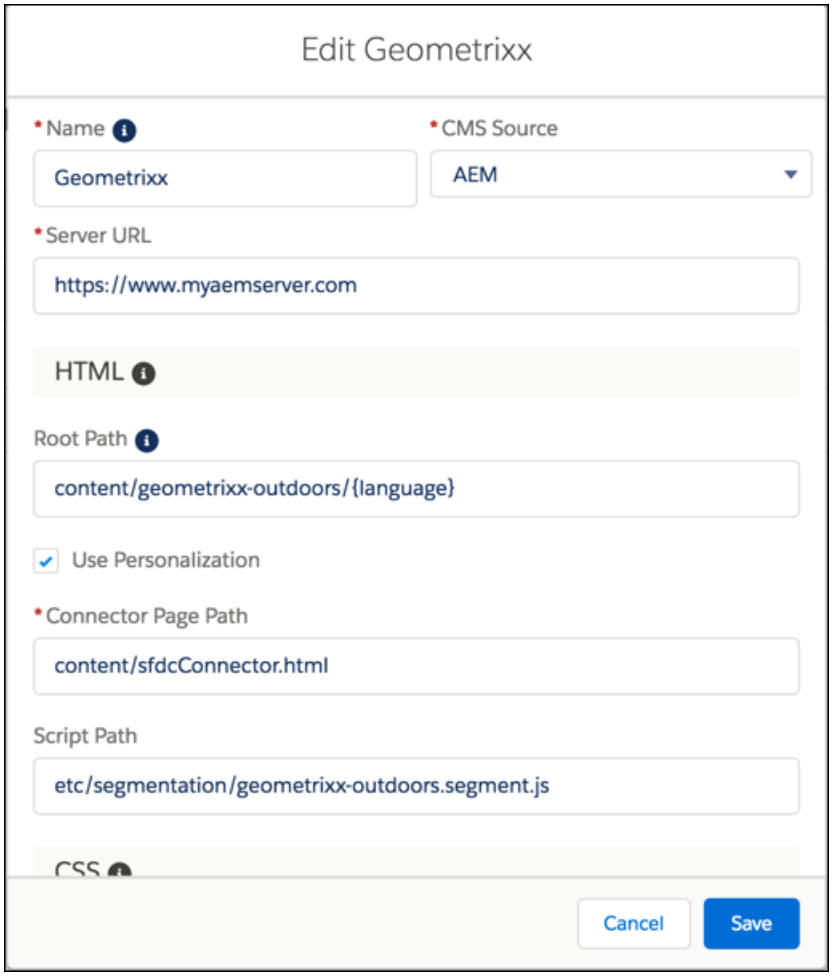
Ready to get started? Let's dig in. (Take a power nap first, if you need to.)

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 85 to create a connector JSP page and expose it through an HTML page in AEM.

Enable Personalization in Your CMS Connection

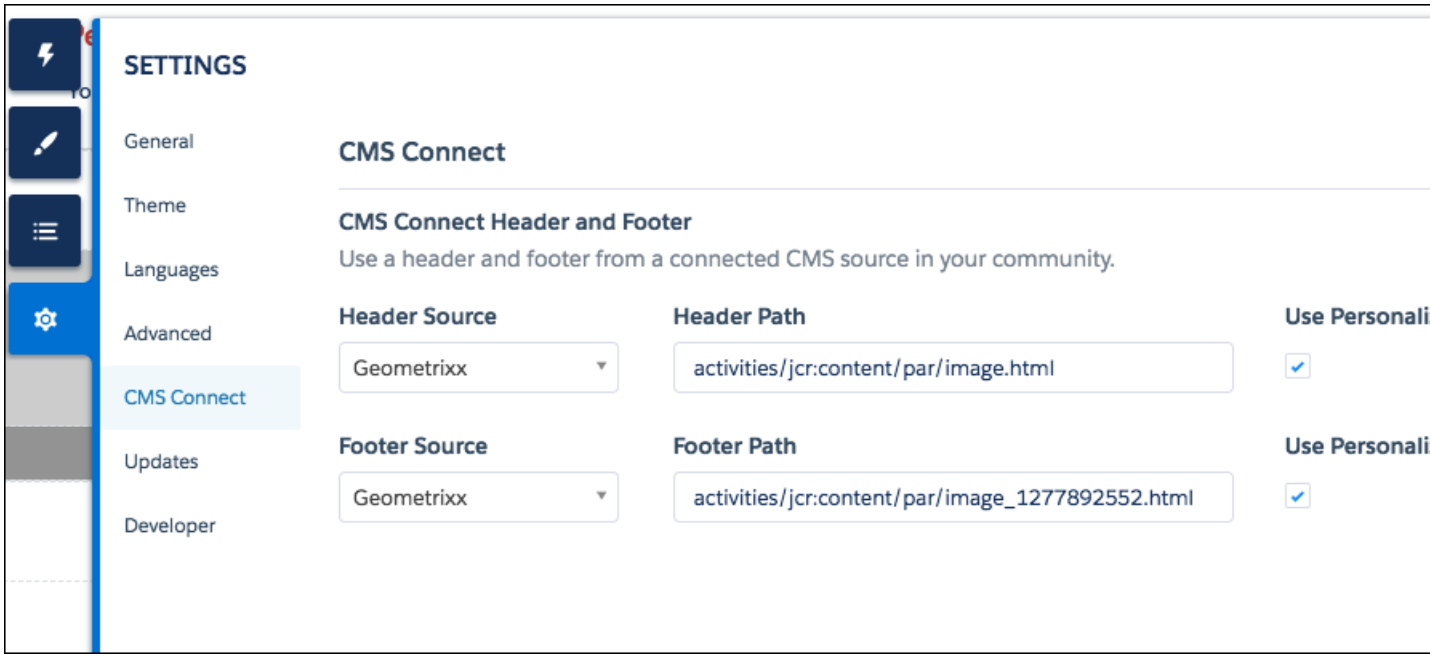
1. Create a CMS connection on page 65 (or edit an existing one on page 74) where you want to host your personalized content.



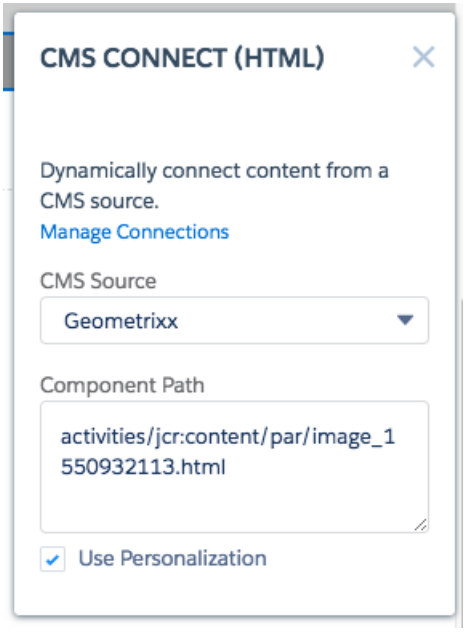
2. Enable **Use Personalization**.
3. In **Connector Page Path**, enter the path to the connector JSP page you installed in AEM.
4. If you want your personalized content to run scripts dynamically, enter the path to your JavaScript file in **Script Path**.
5. Click **Save**.

Enable Personalization in Your Page Components

1. Navigate to Community Builder.
2. 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.)



3. 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.
4. 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.

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.i18n.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("campainName", "");
                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 HTML Content to Your Community

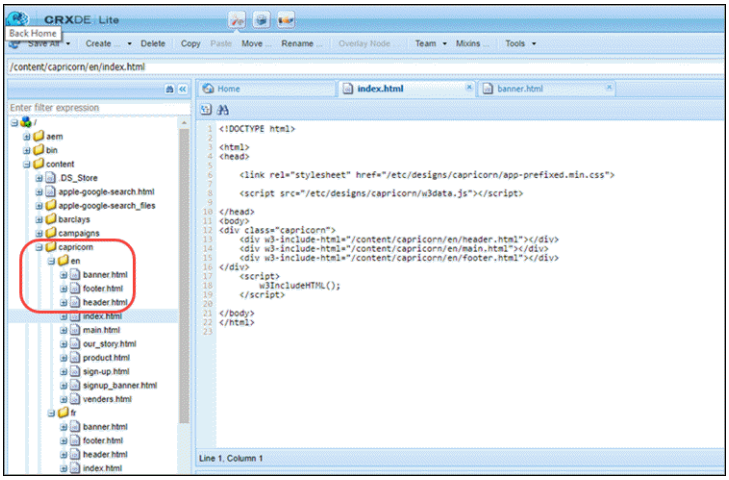
Learn how to display HTML content in your community using CMS Connect. This example brings in an HTML header, footer, and banner content from Adobe Experience Manager (AEM).

Suppose that you use AEM to create content for your site, which is named Capricorn Cafe. To promote the Capricorn Cafe brand in your community, you want to reuse the same look and feel. This example shows how you can create a connection to your CMS and configure settings to retrieve the header and footer. You then add a CMS Connect (HTML) component to include a banner.

Identify Paths to the CMS Content

Before you set up a CMS connection, pinpoint the location of your content, which varies depending on your CMS and setup. Although we test connections to AEM, Drupal, SDL Tridion, Sitecore, and WordPress, a CMS connection isn't provider-specific. You can retrieve and display HTML and JSON content by specifying the endpoints and paths for the data that you want to display.

Examining the content structure on your CMS can help locate the paths. For this example, the content endpoint is `https://capricorn_cafe.com/content/capricorn/en`. In the AEM developer interface, note the path names for the HTML header, footer, and banner.



Set Up a CMS Connection

1. Create a CMS connection on page 65.
2. Under Connection Details, provide this information.
   - Name—*Capricorn*
   - CMS Source—**AEM**
   - Connection Type—**Public**
   - Server URL—*https://capricorn_cafe.com*

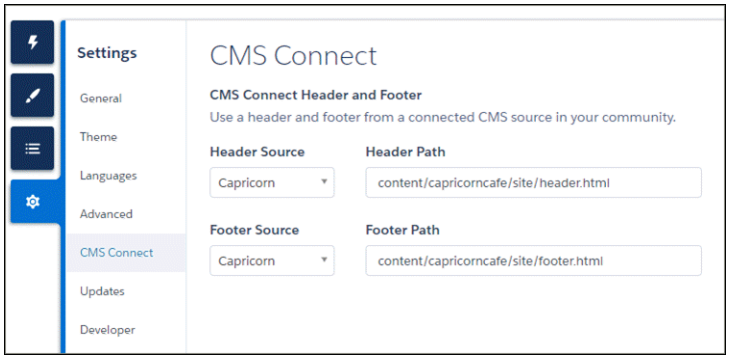
   If your CMS isn't listed, select **Other** as the CMS source. For the server URL, enter a fully qualified domain name for a CMS server that's accessible with HTTPS.
3. (Optional) Set a root path for your CMS content, for example, `content/capricorn/en`. The root path is relative to the server name. It's a shortcut so that you don't have to enter the full path name when you add components later.
4. (Optional) If your content has multiple languages, you can also include a placeholder for the language in the root path, for example, `content/capricorn/{language}`. To configure the connection to support multiple languages, select **Enable language mapping**. Select a language, and enter the CMS language directory that's in AEM. Make sure that each language that you add is enabled in your Community Builder settings.
5. Save your connection settings.

Reuse Header and Footer Content

To bring in the HTML header and footer content, use Community Builder settings, and not CMS Connect (HTML). Defining the header and footer in Settings renders them before other page content.

1. From the Community Builder, go to **Settings**.
2. Click **CMS Connect**.
3. Specify the connection and paths to the header and footer content.
   a. For Header Source, select the connection, and enter a header path, for example, `content/capricorn/en/header.html`.
   b. For Footer Source, select the connection, and enter a footer path, for example, `content/capricorn/en/footer.html`.

   Note: If you defined the root path as `content/capricorn/en`, simply enter `header.html` and `footer.html` for the header and footer paths.

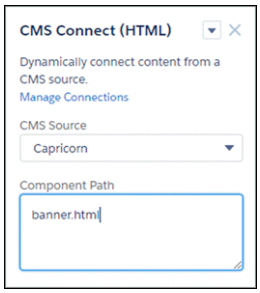


Sometimes pulling in your CMS content requires a little creativity. For example, a SiteFinity master template generally contains other content in addition to a header and footer, all in the same file. In this case, the header content and footer content don't have separate paths. As a workaround, you can construct separate templates: one that contains only the header content and another that contains only the footer content. You can then specify different paths in CMS Connect.

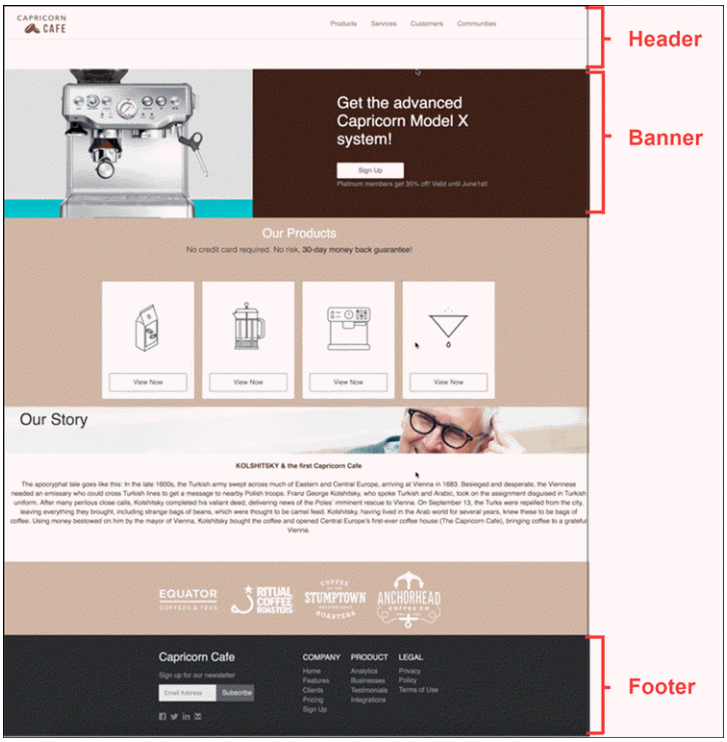
Add Banner Content

To display other HTML content from your CMS, such as the banner, use a CMS Connect (HTML) component from the Community Builder.

1. Drag a CMS Connect (HTML) component to a location on your community page.
2. Configure the component's properties.
   a. For CMS Source, select the connection.
   b. Enter the path to the content, for example, `content/capricorn/en/banner.html`. If you defined the root path in the connection as `content/capricorn/en`, simply enter `banner.html`.



That's it! Your community now displays the site's header, footer, and banner content.



SEE ALSO:

- Set Up a Connection for Your JSON CMS
- Add CMS Header and Footer Components to Your Community
- Add CMS Connect (HTML) Components to Your Community Pages

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 pulls in JSON content from a WordPress CMS into your community.

Let's say you have a WordPress site called Capricorn Cafe. This example shows how to configure a CMS Connect (JSON) component that pulls blog content from that site into your community.

These example endpoints show the location of the JSON content for a content list and a single 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

Suppose that you want to pull in a series of blogs with links to the individual blogs in full.

1. Create a CMS connection on page 65.
2. Under Connection Details, provide the following information.

- Name—*Capricorn*
- CMS Source—**WordPress**
- Connection Type—**Public**
- Server URL—*https://public-api.wordpress.com*

3. Because you're adding JSON content, don't enter an HTML root path, CSS, or scripts. Those sections are for CMS HTML connections.
4. In the JSON section, click **Add JSON**.
   a. For Content Type Name, enter *Blog Feed*.

      Note: A content type name groups lists and items for easy management. You can have up to five content types, each with up to one content item and 10 content lists.

   b. Click **Add content list** and provide the following information.
      - Name—*Blog List*
      - Path—*rest/v1.1/sites/capricorncafeblog.wordpress.com/posts?number={itemsPerPage}&page={pageNumber}*

        Note: The path has two variables used for pagination. *{itemsPerPage}* is the number of items to display on a page. *{pageNumber}* is the current page number. The variable *{offset}* also applies to pagination and specifies a page offset. Variable values are computed dynamically when the CMS Connect (JSON) component is displayed.

      - Node Path—*posts*

        Note: Because a JSON expression is expected in the Node Path, don't enter an @.

   c. To generate a detail page and a link for each blog in the list, make sure that the content type has both a content item and a content list. Click **Add content item** and provide the following information.
      - Name—*Blog Item*
      - Path—*rest/v1.1/sites/capricorncafeblog.wordpress.com/posts/{component}*

        Note: The *{component}* variable adds the component path dynamically. It uses the component path entered for the CMS Connect (JSON) component in the Community Builder.

      - ID Path—*ID*
      - Title Path—*title*

      Note: Because JSON expressions are expected for ID Path and Title Path, don't enter them with an @. JSON expressions are case-sensitive, so ID is different than id. Match what's used in your JSON response.

5. Click **Save**.

Add the Blog Series to Your Page

1. In the Community Builder, drag a CMS Connect (JSON) component to a page.
2. Open the property editor, and configure the connection.

   Note: Use the @ symbol to denote JSON expressions.

   - CMS Source—*Capricorn*
   - JSON Content—**Blog List**

- Component Path—For a content list, leave this field blank.

3. Under Content List Layout, select **Grid**, and provide the following information.
   - Items Per Page—Enter a value, for example, `5`.
   - Columns—Enter a value, for example, `2`.
4. Under Content List Item Layout, select **Card**, and enter properties for the layout.
   - Title—`@title`
   - Author—`@author/name`
   - Published On—`@date`
   - Body—`@content`
   - Image Source—`@featured_image`
5. Under Navigation, set up how the list redirects to a specific blog.
   - Link Text—`Read More`
   - Type—**Community Page**
6. Click **Save** in the property editor. After saving, you see the name created for the detail page, for example, Capricorn-Blog_Feed. The detail page shows the full blog item in the series.

   Note: To change the name or URL of the detail page, locate the page in the Pages menu in the top toolbar. Then modify its properties.

Your community page now displays the blog series using your CMS content. After you publish the changes, you can click the Read More link under a blog item in the series. The link takes you to a detail page generated dynamically for the blog.

After you publish your community, you can change your CMS connection settings. For example, suppose that you modify a content list path to point to news items instead of blog items. Because CMS connections are cached for the guest user, republishing your community allows the guest to see the update without a delay.

Add a Single Blog Item to a Page

But what if all you need is a single blog or news item that you want to feature on your community page?

1. Create a CMS connection. In this example, let's reuse the same connection.
2. In the Community Builder, drag a CMS Connect (JSON) component to a page.
3. Open the property editor, and configure the component.
   - CMS Source—`Capricorn`
   - JSON Content—**Blog Item**
   - Component Path—Enter the ID for the blog, for example, `38`

     Note: When you specify the component path for a content item, you can also use a dynamic variable, such as `{!id}`. When you click a Read More link, for example, an ID value replaces the variable.
4. Under Content List Item Layout, select **Detail**, and enter properties for the item's layout.
   - Title—`@title`
   - Author—`@author/name`
   - Published On—`@date`

- Body—`@content`
- Image Source—`@featured_image`

5. Click **Save**.

Congratulations! Your community page is now configured to display your blog item.

SEE ALSO:

Set Up a Connection for Your JSON CMS

CMS Connect (JSON) Expressions

CHAPTER 8 Report on Deflections: The Deflection Signals Framework

In this chapter ...

A `lightningcommunity:deflectionSignal` event is fired in a community or portal when a user is trying to solve an issue, such as a customer service problem, and then views a deflection item that potentially provides a solution.

For example, let's say a user is filling out a form to create a customer case, and then sees a useful article on the page. After clicking the article, the user finds it helpful and decides that creating a case is unnecessary. A `lightningcommunity:deflectionSignal` event is then fired with information about the user's interaction with the article, which is reported as a successful deflection because the user didn't create a case.

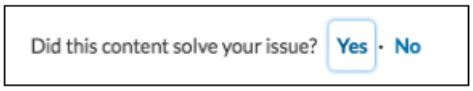
You can report on these events through custom report types with the target object Community Case Deflection Metrics. The signal appears in reports as either a Successful Deflection, Failed Deflection, or Potential Deflection.

SEE ALSO:

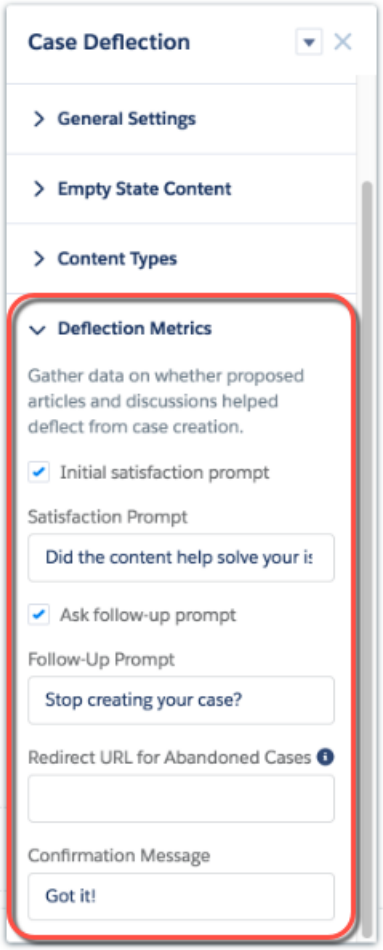
Case Create Deflection Signal

Case Create Deflection Signal

This `lightningcommunity:deflectionSignal` event is fired in a Lightning community when a user is deflected away from creating a customer case. After viewing an article or discussion in a community, the user is asked if the interaction was helpful, and whether they want to stop creating their case.



You can configure the Case Deflection component to fire this event automatically using the component's Deflection Metrics properties in Community Builder. The Case Deflection component works together with the Contact Support Form to register deflection interactions.



Attributes

The `sourceType` for deflection signals from the Case Deflection component is `caseCreateDeflectionModal`.

The `source` is what the user has typed into the subject field or the description of the Case Create Form. The `destination` is the ID of the Article or Discussion deflection item.

The `payload` is a JavaScript object key-value mapping. The following properties are used for this type of signal.

Payload Property	Type	Description	Supported Values	Required
`deflectionAnswer`	string	The user's answer to the first question, asking whether the deflection item was helpful.	• `YES` • `NO` • `null`—the user didn't vote	No
`confirmationAnswer`	string	The user's answer to the second question, asking whether they wish to stop creating a case.	• `YES` • `NO` • `null`—the user didn't vote	No
`state`	string	The state the popup window was last left in before it was closed.	• ~~MeasureDeflectionState~~—the user didn't answer the first question • ~~ConfirmationQuestionState~~—the user didn't answer the second question • ~~ConfirmationMessageState~~—the user answered both questions	No
`caseCreated`	boolean	Indicates whether the user created a case.	• `true`—the user created a case • `false`—the user didn't create a case	No

Examples

Custom Lightning components can listen to this system event and handle it as required, such as starting another process if the user didn't find the content helpful.

Here's a sample component that listens to the system event.

```
<aura:component implements="forceCommunity:availableForAllPageTypes">
    <aura:attribute name="message" type="String" required="false"/>
    <aura:handler event="lightningcommunity:deflectionSignal" action="{!c.handleSignal}"/>

    <lightning:formattedText value="{!v.message}"/>
</aura:component>
```

This client-side controller example handles the system event and checks for failed case deflections (that is, interactions where the user didn't find the deflection item helpful).

```
({
    handleSignal: function(component, event, helper) {
        var signal = event.getParams() || {},
            sourceType = signal.sourceType,
            payload = signal.payload;
        // Process case create deflection signals
        if (sourceType && sourceType === "caseCreateDeflectionModal") {
            if (payload && payload.deflectionAnswer === "NO") {
                component.set("v.message", "Sorry you didn't find that helpful.");
            }
            if (payload && payload.caseCreated === true) {
                component.set("v.message", "We Apologize For The Inconvenience. We'll get
 in touch with you shortly about your case.");
            }
        }
    }
})
```

Custom Lightning components that act as case create forms and case deflection components can also fire this event. Given valid parameters, the event is automatically handled and processed for reporting. This example fires a `lightningcommunity:deflectionSignal` event with values from the component attributes.

```
fireCaseDeflectionSignal : function(component, shouldSubmitSourceTypeSignals) {
    var evt = $A.get("e.lightningcommunity:deflectionSignal");
    evt.setParams({
        sourceType: "caseCreateDeflectionModal",
        source: cmp.get("v.deflectionTerm"),
        destinationType: component.get("v.deflectionEntityType"),
        destination: component.get("v.deflectionEntityId"),
        payload: {
            deflectionAnswer: component.get("v.deflectionAnswer"),
            confirmationAnswer: component.get("v.confirmationAnswer"),
            state: component.get("v.deflectionState"),
            caseCreated: component.get("v.caseCreated")
        },
        shouldSubmitSourceTypeSignals: shouldSubmitSourceTypeSignals
    });
    evt.fire();
}
```

A user can successively view multiple deflection items before ultimately deciding whether to create or abandon a case. Each view fires a `lightningcommunity:deflectionSignal` event. If you want to process all the events as a single batch, set `shouldSubmitSourceTypeSignals=true` for the final event in which the user abandons or creates the case. This example fires the last deflection signal event, based on whether the case was created or not.

```
fireCaseCreatedSignal : function(component, caseCreated) {
    // Send all accumulated signals to the server to be processed
    var evt = $A.get("e.lightningcommunity:deflectionSignal");
    evt.setParams({
        sourceType: "caseCreateDeflectionModal",
        payload: {
            caseCreated: caseCreated
```

```
        },
        shouldSubmitSourceTypeSignals: true
    });
    evt.fire();
}
```

CHAPTER 9 Deploy a Community from Sandbox to Production

In this chapter ...

We recommend creating, customizing, and testing your community in a test environment first, before you deploy it to your production org.

When testing is complete, you can use change sets or the Metadata API to migrate your community from one org to another. Both options are ideal for building and testing a community in a test environment, such as sandbox, and then deploying to another org, such as production.

- **Change sets**—If you're more comfortable working with point-and-click tools, then change sets are your deployment friend.

  Change sets represent sets of customizations in your org (or metadata components) that you can deploy to a connected org.

- **Metadata API**—Up to speed on the Metadata API and more comfortable in the world of code? Then use the Metadata API to deploy changes programmatically.

  The Metadata API lets you retrieve, deploy, create, update, or delete customization information for your org, such as communities, custom object definitions, and page layouts.

EDITIONS

Available in: Salesforce Classic (not available in all orgs) and Lightning Experience

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

Deploy 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. From Setup, select **Inbound Change Sets** and find the change set that you uploaded from your source org.
8. 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.

9. Manually reconfigure any unsupported items in the target org community.

EDITIONS

Available in: Salesforce Classic (not available in all orgs) 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

  Note: If a user requires access to metadata but not to data, you can enable the Modify Metadata permission (beta) to give the access the user needs without providing access to org data. See "Modify Metadata Permission (Beta)" in Salesforce Help.

To use outbound change sets:

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

10. 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 Deploying Communities with Change Sets
- Change Sets Best Practices
- Upload Outbound Change Sets
- Deploy Inbound Change Sets

Considerations for Deploying 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 (not available in all orgs) and Lightning Experience

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

General

- 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 you also update the template 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
- Dashboards and engagement
- 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 **Select which login options to display** option in the Login 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

Deploy Your Community with Change Sets

Deploy Your Community with the Metadata API

Use the Metadata API to move your community from one Salesforce org to another. Set up and test your community in your test environment, and then retrieve the community's data and deploy it to your production org.

You can use the Metadata API to move Lightning communities and Salesforce Tabs + Visualforce communities.

The following metadata types combine to define a community. To successfully migrate a community, use the Metadata API `retrieve` call to retrieve XML file representations of your org's components.

- `Network`—Represents a community. Contains administration settings, such as page override, email, and membership configurations.
- `CustomSite`—Contains the domain and page setting information, including `indexPage`, `siteAdmin`, and URL definitions.
- `SiteDotCom`—Contains the community site layout.

EDITIONS

Available in: Salesforce Classic (not available in all orgs) and Lightning Experience

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

For additional information on these metadata types and instructions on migrating data, see the *Metadata API Developer Guide* and the *Ant Migration Tool Guide*.

Tips and Considerations

- Before migrating data to another org, enable Communities in the destination org and enter the same domain name that you used in your sandbox org to avoid getting an error.
- For each community, the network component has a unique name and URL path prefix. When you retrieve the network component, the generated XML file name is based on the name of the network. When migrating, the API looks at the file name and if it exists, updates the community. If it doesn't exist, the API creates a community. If someone changes the community name in the sandbox and then tries to migrate, they see an error because the API is trying to create a community with the existing path prefix.
- Examine the XML file for `CustomSite` to make sure that all dependencies are brought over. If any are missing, explicitly state them in the XML file.
- In addition to the `Network`, `CustomSite`, and `SiteDotCom` components, include all the other components required by your community, such as custom objects, custom fields, custom Lightning components, and Apex classes.
- If you rename a community in **Administration** > **Settings**, make sure that the source and target communities have matching values for the `picassoSite` and `site` attributes in the `Network` component.
- If there are any changes to the guest user profile, include the profile as part of the community migration.
- When you migrate user profiles, users are added to the community in the production org. Emails are then sent to members in the same way as for any new community.

Sample Template

The following sample contains all of the fields that you can migrate through the Metadata API.

```
<?xml version="1.0" encoding="UTF-8"?>
<Network xmlns="http://soap.sforce.com/2006/04/metadata">
    <allowInternalUserLogin>true</allowInternalUserLogin>
    <allowMembersToFlag>true</allowMembersToFlag>
    <allowedExtensions>txt,png,jpg,jpeg,pdf,doc,csv</allowedExtensions>
   <caseCommentEmailTemplate>unfiled$public/ContactFollowUpSAMPLE</caseCommentEmailTemplate>


<changePasswordTemplate>unfiled$public/CommunityChangePasswordEmailTemplate</changePasswordTemplate>

    </communityRoles>
    <disableReputationRecordConversations>true</disableReputationRecordConversations>
    <emailSenderAddress>admin@myorg.com</emailSenderAddress>
    <emailSenderName>MyCommunity</emailSenderName>
    <enableCustomVFErrorPageOverrides>true</enableCustomVFErrorPageOverrides>
    <enableDirectMessages>true</enableDirectMessages>
    <enableGuestChatter>true</enableGuestChatter>
    <enableGuestFileAccess>false</enableGuestFileAccess>
    <enableInvitation>false</enableInvitation>
    <enableKnowledgeable>true</enableKnowledgeable>
    <enableNicknameDisplay>true</enableNicknameDisplay>
    <enablePrivateMessages>false</enablePrivateMessages>
    <enableReputation>true</enableReputation>
    <enableShowAllNetworkSettings>true</enableShowAllNetworkSettings>
    <enableSiteAsContainer>true</enableSiteAsContainer>
```

```
    <enableTalkingAboutStats>true</enableTalkingAboutStats>
    <enableTopicAssignmentRules>true</enableTopicAssignmentRules>
    <enableTopicSuggestions>true</enableTopicSuggestions>
    <enableUpDownVote>true</enableUpDownVote>

<forgotPasswordTemplate>unfiled$public/CommunityForgotPasswordEmailTemplate</forgotPasswordTemplate>

    <gatherCustomerSentimentData>false</gatherCustomerSentimentData>
    <lockoutTemplate>unfiled$public/CommunityLockoutEmailTemplate</lockoutTemplate>
    <maxFileSizeKb>51200</maxFileSizeKb>
    <navigationLinkSet>
        <navigationMenuItem>
            <label>Topics</label>
            <position>0</position>
            <publiclyAvailable>true</publiclyAvailable>
            <target>ShowMoreTopics</target>
            <type>NavigationalTopic</type>
        </navigationMenuItem>
    </navigationLinkSet>
    <networkMemberGroups>
        <permissionSet>MyCommunity_Permissions</permissionSet>
        <profile>Admin</profile>
    </networkMemberGroups>
    <networkPageOverrides>
       <changePasswordPageOverrideSetting>VisualForce</changePasswordPageOverrideSetting>

        <forgotPasswordPageOverrideSetting>Designer</forgotPasswordPageOverrideSetting>
        <homePageOverrideSetting>Designer</homePageOverrideSetting>
        <loginPageOverrideSetting>Designer</loginPageOverrideSetting>
        <selfRegProfilePageOverrideSetting>Designer</selfRegProfilePageOverrideSetting>
    </networkPageOverrides>
    <picassoSite>MyCommunity1</picassoSite>
    <selfRegistration>true</selfRegistration>
    <sendWelcomeEmail>true</sendWelcomeEmail>
    <site>MyCommunity</site>
    <status>Live</status>
    <tabs>
        <defaultTab>home</defaultTab>
        <standardTab>Chatter</standardTab>
    </tabs>
    <urlPathPrefix>mycommunity</urlPathPrefix>
    <welcomeTemplate>unfiled$public/CommunityWelcomeEmailTemplate</welcomeTemplate>
</Network>
```

Sample `package.xml` Manifest File

A manifest file defines the components that you're trying to retrieve. The following sample shows a `package.xml` manifest file for retrieving all the components of a community.

```
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
    <types>
        <members>*</members>
```

```
        <name>Network</name>
    </types>
    <types>
        <members>*</members>
        <name>CustomSite</name>
    </types>
    <types>
        <members>*</members>
        <name>SiteDotCom</name>
    </types>
    <types>
        <members>*</members>
        <name>CustomTab</name>
    </types>
    <types>
        <members>*</members>
        <name>CustomObject</name>
    </types>
    <types>
        <members>*</members>
        <name>ApexClass</name>
    </types>
    <types>
        <members>*</members>
        <name>ApexPage</name>
    </types>
    <types>
        <members>*</members>
        <name>ApexComponent</name>
    </types>
    <types>
        <members>*</members>
        <name>Portal</name>
    </types>
    <types>
        <members>*</members>
        <name>Profile</name>
    </types>
    <types>
        <members>*</members>
        <name>Document</name>
    </types>
    <version>43.0</version>
</Package>
```

SEE ALSO:

Deploy Your Community with Change Sets

INDEX