Adobe

Student Workbook

Adobe Experience Manager
Develop Websites and Components

Develop Websites and Components in AEM

5/8/2023

Contents

Module 1 6

Introduction to Content Rendering 6

- JCR Nodes, Properties, and Namespaces 7
- JCR Folder Structure 9
- Structure of a Component 10
- Exercise 1: Create an HTL Component 11
- Exercise 2: Create Content to Render the Component 13
- Sling Resolution Process 15
- Exercise 3: Search for a Rendering Script 20
- Exercise 4: Use the Authoring Interface to Add Content 24
- Dialogs 27
- Exercise 5: Create an Edit Dialog 28
- Exercise 6: Customize Selectors 33
- References 36

Module 2 37

Introduction to HTML Template Language 37

- HTL 38
- HTL: Goals 39
- HTL Syntax 40
- Exercise 1: Render Page Content by Using AEM Global Objects 42
- Exercise 2: Render Page Content by Using HTL Attributes 45
- References 47

Module 3 48

Getting Started with Sites Components 48

- Components 49
- Core Components Overview 50
- Exercise 1: Investigate Components 51
- Exercise 2: Install Latest Core Components (Optional AEM 6.5.X only) 56
- Experience Manager Features Enabled by Core Components 60
- References 63

Module 4 64

Develop Designs for Components 64

Client-Side Libraries 65
Organizing Client-Side Libraries 67
Referencing Client-Side Libraries 68
Exercise 1: Investigate a Client Library 69
Exercise 2: Investigate the ui.frontend Maven Module 72
Exercise 3: Add Content to a Page 76
Exercise 4: Set up Webpack for Development 80
Exercise 5: Create a Design for a Component 83
Exercise 6: Add a Page Style Using the Style System 88
Exercise 7: Investigate the Page Component Inclusions 95
Exercise 8: Include Client Libraries via Template Policies 99
References 103

Module 5 104

Extend Core Components104

Core vs. Extend vs. Custom 105
Exercise 1: Extend a Core Component 109
Exercise 2: Customize the Component Dialog 114
Exercise 3: Enable Authoring Actions Using cq:editConfig 122
Exercise 4: Add a Default Design to a Component 126
(Optional) Exercise 5: Create a Sling Model for Headless Output 130
References 133

Module 6 134

Implement Custom Components 134

Features of Components 135
Exercise 1: Create a Custom Component 138
Exercise 2: Create a Custom Dialog with Validation 143
Exercise 3: Create a Content Policy to Control Component Behavior 152
Exercise 4: Add a Default Design and Optional Styles 160
Exercise 5: Add Styles to a Content Policy 164
Exercise 6: Create Translation Dictionaries 169
(Optional) Exercise 7: Create Additional Supported Languages 173
Exercise 8: Add a Sling Model as Business Logic 177
Adobe Client Datalayer 182
(Optional) Exercise 9: Investigate the Core Component Use of the Adobe Client Data Layer 184
Exercise 10: Enable a Website for the Adobe Client Data Layer 187
Exercise 11: Enable a Custom Component to Populate the Adobe Client Data Layer 190
(Optional) Exercise 12: Enable a Clickable Event in the Adobe Client Data Layer 193
(Optional) Exercise 13: Examine Data Layer Enablement in the Sling Model for a Component 195
References 196

Module 7 197

Work with Page Templates 197

Templates 198
Creating Editable Templates 200
Exercise 1: Create and Investigate a Page Template 204
Creating Pages from Editable Templates 213
Exercise 2: Finish Creating Initial Templates 215
Exercise 3: Build Out the Website 221
Exercise 4: Author the Header and Footer 227
Exercise 5: Create a Template Type with Header and Footer 237

Introduction to Content Rendering

Introduction

Adobe Experience Manager (AEM) is a content management system that helps organizations build custom websites and apps across different customer touch points. To better utilize AEM features, you need to understand the key concepts such as Java Content Repository (JCR), base page component, Sling resolution, and the inheritance process.

Objectives

After completing this module, you will be able to:

- Explain JCR nodes, properties, and namespaces
- Explain the JCR folder structure
- Describe the structure of a component
- Create a component
- Create an edit dialog
- Explain the Sling resolution process
- Search for a rendering script
- Customize selectors

JCR Nodes, Properties, and Namespaces

The JCR has a hierarchical tree structure with two items, nodes and properties.

Nodes

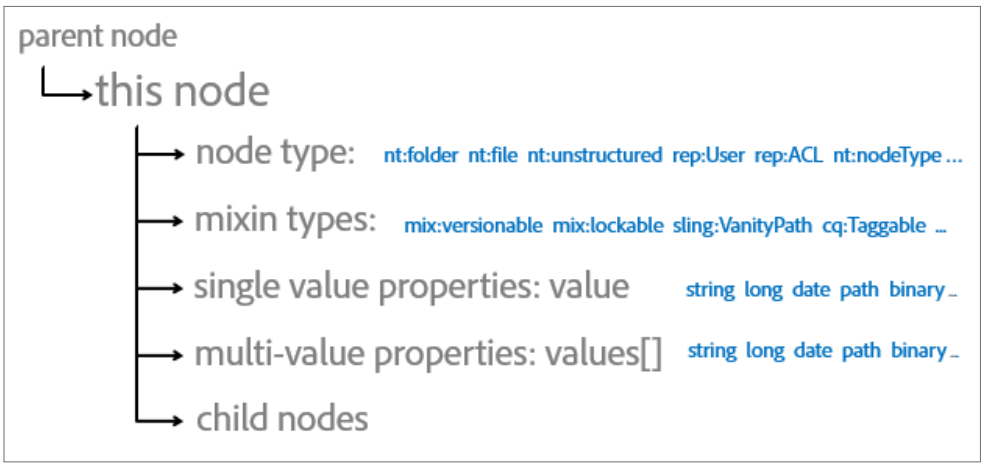
Nodes provide the structure and properties to store the data. The nodes of JCR can have:

- Parent and child nodes that are represented with paths.
- A type that sets the rules about the kind of properties and child nodes that a node can or must have. For example, a node of type cq:page must have a child node jcr:content of type cq:PageContent
- Any number of properties.
- Zero or more mixin types that specify additional properties and child nodes that a node must or may have. The common mixin types include mix:versionable and mix:referenceable.

Properties

The properties of a node have a:

- Name and a value
- Single or multi-valued property. A multi-valued property is notated and behaves like an array (values []).



Namespaces

All nodes and JCR properties are stored in different namespaces. The common namespaces are:

- jcr: Basic data storage (part of jcr spec)
- nt: Foundation node types (part of jcr spec)
- rep: Repository internals (part of jcr spec)
- mix: Standard mixin node types (part of jcr spec)
- sling: Added by Sling framework
- cq: Added by the AEM application

The following table describes the common node types used in AEM:

Node type	Description
nt:file	Represents a file in a filesystem.
nt:folder	Represents a folder in a filesystem.
nt:unstructured	Supports any combination of child nodes and properties. It also supports client-orderable child nodes, stores unstructured content, and is commonly used for touch UI dialog boxes.

The following table describes the common AEM node types:

Node type	Description
cq:Page	Stores the content and properties for a page in a website.
cq:Template	Defines a template used to create pages.
cq:ClientLibraryFolder	Defines a library of client-side JavaScript CSS.
cq:EditConfig	Defines the editing configuration for a component including drag-and-drop and in-place editing.
cq:InplaceEditingConfig	Defines an in-place editing configuration for a component. It is a child node of cq:EditConfig.

JCR Folder Structure

You can view the folder structure of JCR in CRXDE Lite. The following table describes the folder structure within the repository:

Folder	Description
/apps	Contains all immutable project code such as components, overlays, client libraries, bundles, i18n translations, and static templates created by an organization.
/conf	Any configurations in this location could be modified at runtime with an application console..
/content	Contains content created by authors using typical AEM consoles.
/etc	Legacy location used only for backwards compatibility with older applications in AEM..
/home	Contains AEM users and group information.
/libs	Contains the libraries and definitions that belong to the core of AEM. The subfolders in /libs represent the out-of-the-box AEM features.
/oak:index	Contains Jackrabbit Oak index definitions. Each node specifies the details of one index. The standard indexes for the AEM application are visible and help create additional custom indexes.
/system	Is used by Apache Oak only.
/tmp	Serves as a temporary working area.
/var	Contains the files that are updated by the system, such as audit logs, statistics, and event-handling. The subfolder /var/classes contains the Java servlets in source and compiled forms that are generated from the components scripts.

Caution: You may need to change the JCR folder structure during website development. However, you should fully understand the implications of any changes you make. Most changes will occur in /apps during development.

Structure of a Component

Common elements of component are:

- Root component node
- Component node properties
- Component child nodes for special functions

Root component node

It is the hierarchy node of the component and is represented as node `<mycomponent>` (type `cq:Component`).

Component node properties

The vital properties of a component are:

- jcr:title: Is the component title. For example, this property is used as a label when the component is listed in the components browser.
- jcr:description: Is the description for the component. You can use this property as a tooltip in the components browser.
- cq:icon: Is the string property pointing to a standard icon in the Coral UI library, which is displayed in the component browser.

Component child nodes for special functions

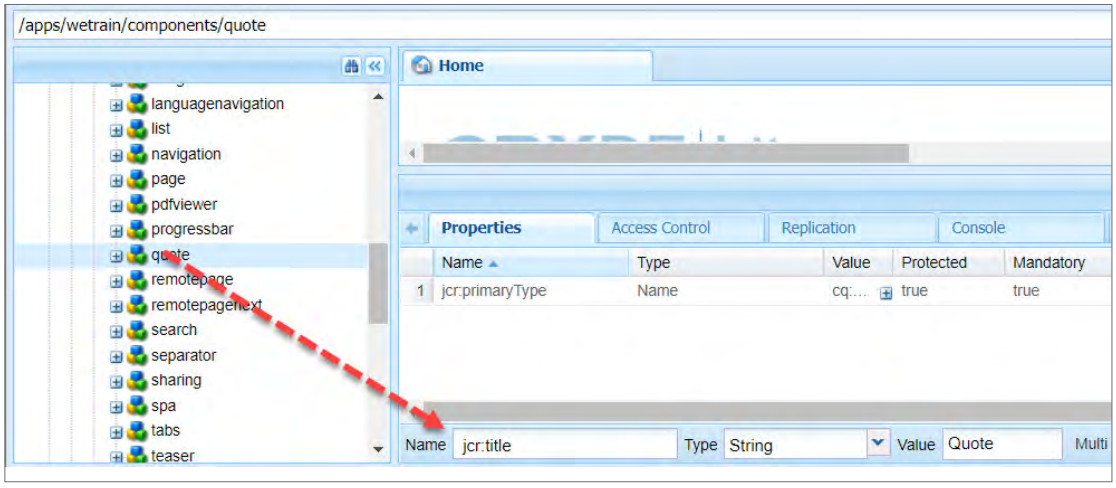
The vital child nodes of a component are:

- cq:editConfig (cq:EditConfig): Defines the edit properties of the component and enables the component to appear in the components browser.
- cq:childEditConfig (cq:EditConfig): Controls the author UI aspects for child components that do not define their own cq:editConfig.
- cq:dialog (nt:unstructured): Is the dialog for a component and defines the interface, which enables the user to configure the component and/or edit content.
- cq:design_dialog (nt:unstructured): Helps edit the design of a component.

Exercise 1: Create an HTL Component

In this exercise, you will create a simple component that can be rendered using a Sling resource.

1. In **CRXDE Lite**, navigate to **/apps/wetrain/components/**.
2. Right-click the **components** folder and click **Create** > **Create Node**. The **Create Node** dialog box opens.
3. In the **Name** box, type **quote**.
4. From the **Type** dropdown menu, select **cq:Component**.
5. Click **OK**. A node of component type is created.
6. Click **Save All** to save the changes.
7. Select the **quote** component you created to add properties:
   a. In the **Name** box, type **jcr:title**.
   b. In the **Type** box, retain **String**.
   c. In the **Value** box, type **Quote**.
   d. Click **Add**, as shown. The property is added.



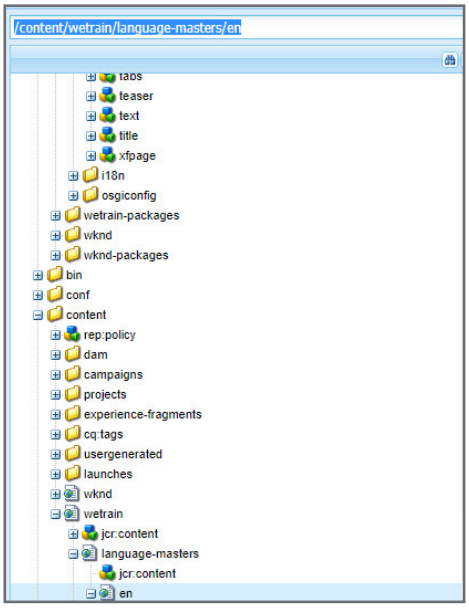
8. Click **Save All** to save the changes.

9. Right-click the **quote** component and click **Create** > **Create File**. The **Create File** dialog box opens.
10. In the **Name** box, type **quote.html** and click **OK**. The file is created.
11. Click **Save All** to save the changes. Notice that the **quote.html** editor opens on a separate tab on the right in **CRXDE Lite**.
12. In the **Exercise_Files-DWC** folder provided to you by your instructor, navigate to the **ui.apps/src/main/content/jcr_root/apps/wetrain/components/quote** folder.
13. Open the **start-quote.html** file in Notepad++ (or any editor of your choice), copy the content of the file and paste it in the **quote.html** editor in **CRXDE Lite**.
14. Click **Save All** to save the changes.

Exercise 2: Create Content to Render the Component

In this exercise, you will create a content node that renders the component. This teaches the basics of rendering the code (the component) with content. Creating content through CRXDE Lite is not typical though. Typically, authors create the content using the AEM Sites console. You saw an example of this with the quote component, adding it to a page. Use CRXDE Lite to quickly create a testing content node.

1. In **CRXDE Lite**, navigate to the **/content/wetrain/language-masters/en** folder, as shown:

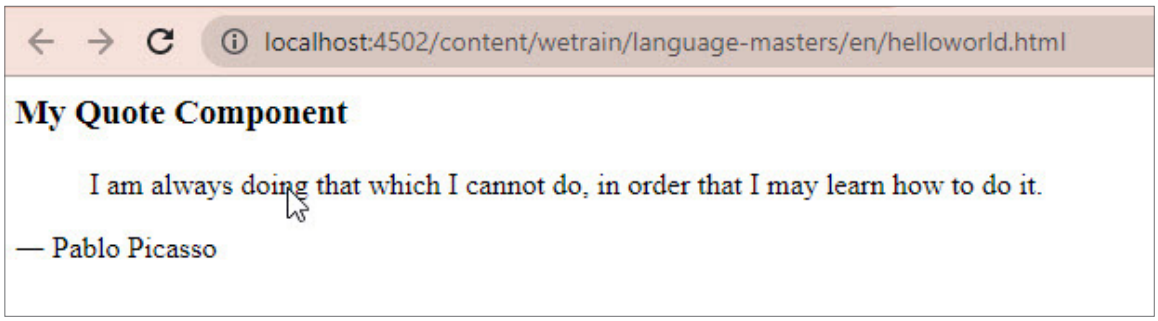


2. Right-click the **/en** folder and click **Create** > **Create Node**. The **Create Node** dialog box opens.
3. Enter the following details:
   a. **Name**: **helloworld**
   b. **Type**: **cq:Page**
4. Click **OK**. The **helloworld** node is created.
5. Click **Save All**.

6. Right-click the **helloworld** node you created in the previous step and click **Create** > **Create Node**. The **Create Node** dialog box opens.
7. Enter the following details:
   a. **Name**: **jcr:content**
   b. **Type**: **cq:PageContent**
8. Click **OK**. The **jcr:content** node is created.
9. Click **Save All**.
10. Select the **jcr:content** node and add the following property on the **Properties** tab:

Name	Type	Value
sling:resourceType	String	wetrain/components/quote

11. Click **Add**. The property is added to the node.
12. Click **Save All**.
13. In a browser, open http://localhost:4502/content/wetrain/language-masters/en/helloworld.html
14. Observe how the property `sling:resourceType` renders the content from **quote.html** on the **helloworld.html** page:



Sling Resolution Process

Apache Sling is resource-oriented, and all resources are maintained in the form of a virtual tree. A resource is usually mapped to a JCR node. However, you can also map the resource to a file system or a database. The common properties that a resource can have are Path, Name, and Resource Type.

Resource First Request Processing

A request URL is first resolved to a resource and based on the resource, Sling selects the servlet or the script to handle the request.

The following table lists the differences between a traditional framework and a Sling framework request processing:

Traditional web application framework	Sling framework
Selects the servlet or controller based on the request URL	Places data in the center
Loads data from the database to render the result	Uses the request URL to resolve the data to process

Processing Requests: Steps

Each content item in JCR is exposed as an HTTP resource. After the content is determined, the script or the servlet to be used to handle the request is determined through the following:

- The properties of the content item
- The HTTP method used to make the request
- The simple naming convention within the URL that provides the secondary information

The steps involved in resolving a URL request are:

1. Decompose the URL.
2. Search for a servlet or a vanity URL redirect.
3. Search for a node indicated by the URL.
4. Resolve the resource.
5. Resolve the rendering script/servlet.
6. Create a rendering chain.
7. Invoke a rendering chain.

Decomposing the URL

Consider the following URL:
http://myhost/tools/spy.printable.a4.html/a/b?x=12

This URL can be decomposed into the following components:

Protocol	Host	Content Path	Selector(s)	Extension		Suffix		Param(s)
http://	myhost	tools/spy	.printable.a4	.html	/	a/b	?	x=12

where,

- Protocol: Is the Hypertext transfer protocol (HTTP)
- Host: Is the name of the website
- Content path: Is the path specifying the content to be rendered
- Selector(s): Is used for alternative methods of rendering the content
- Extension: Is the content format that also specifies the script to be used for rendering
- Suffix: Is used to specify additional information
- Param(s): Are any parameters required for dynamic content

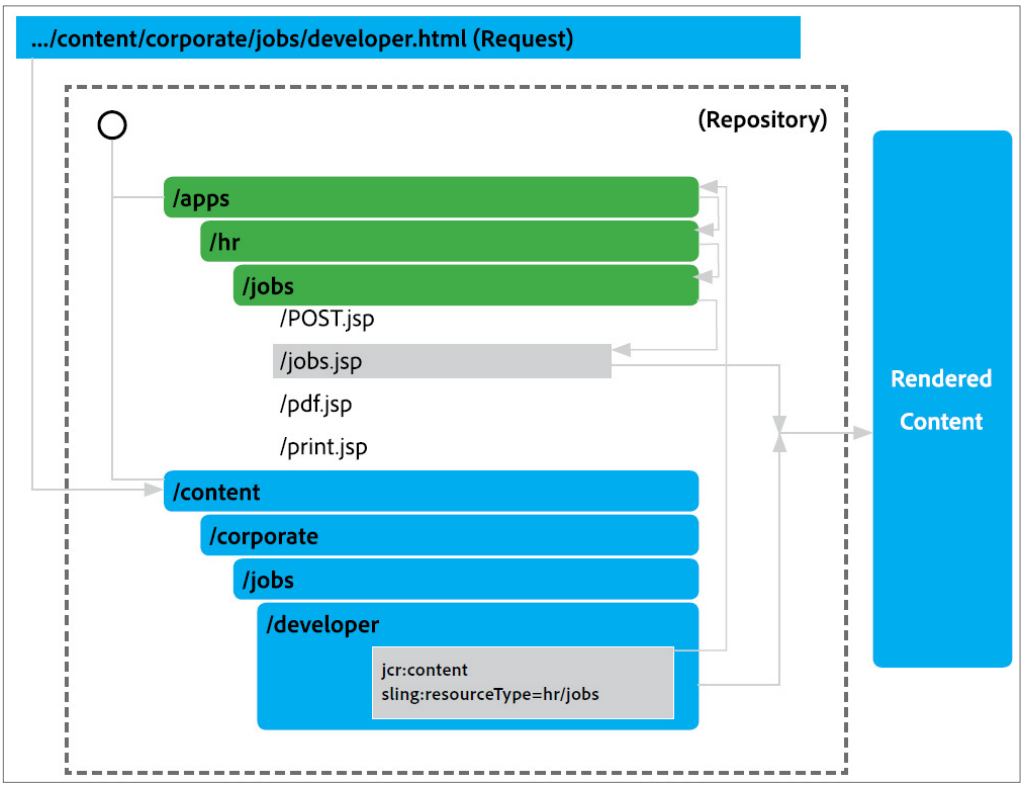
Resolving Requests to Resources

Example 1: URL: http://myhost/tools/spy.html
After the URL is decomposed, the content node is located from the content path. This node is identified as the resource and performs the following steps to map to the request:

1. The Sling Resource Resolver process first looks for a redirection rule such as a vanity URL or a servlet. If the rule/servlet does not exist or is not found, the script resolution process begins.
2. As part of the script resolution process, Sling searches for the spy node.

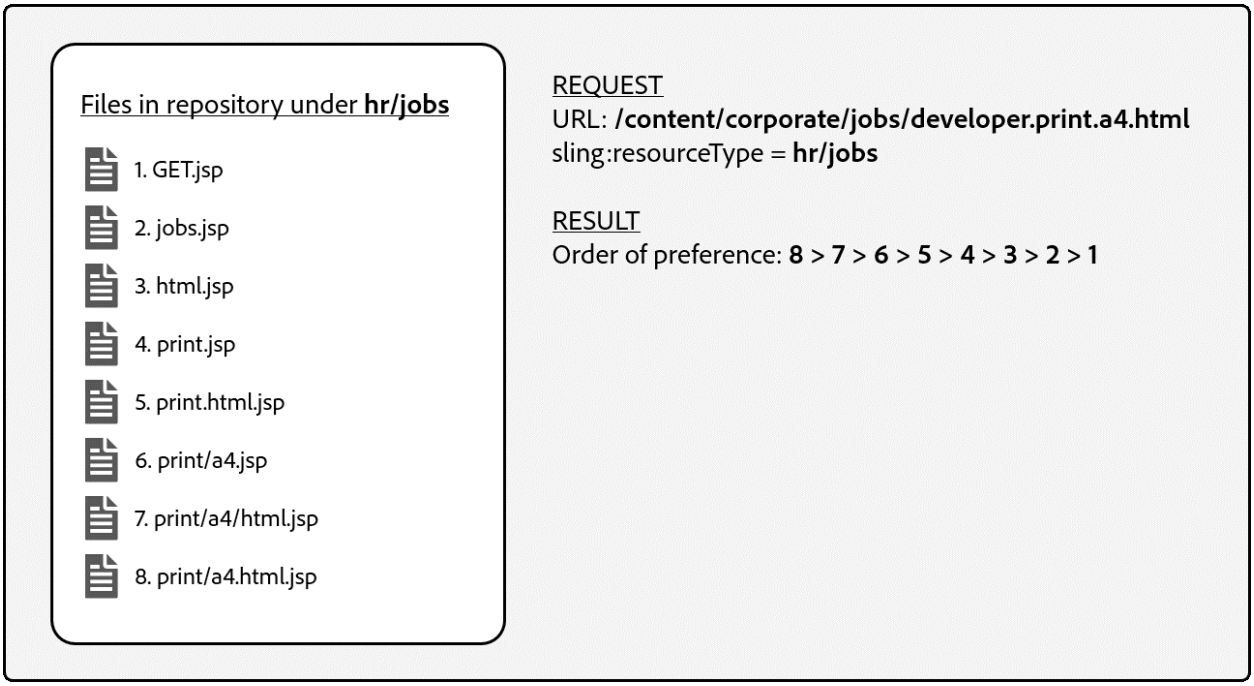
3. If a node is found, the sling resource type for that node is extracted, and used to locate the script to be used for rendering the content.
4. If no node is found, Sling will return the http code 404 (Not Found).

Example 2: Consider the URL request, http://myhost/content/corporate/jobs/developer.html and the corresponding diagram. Notice how the properties of the developer node are used to provide the rendered content:



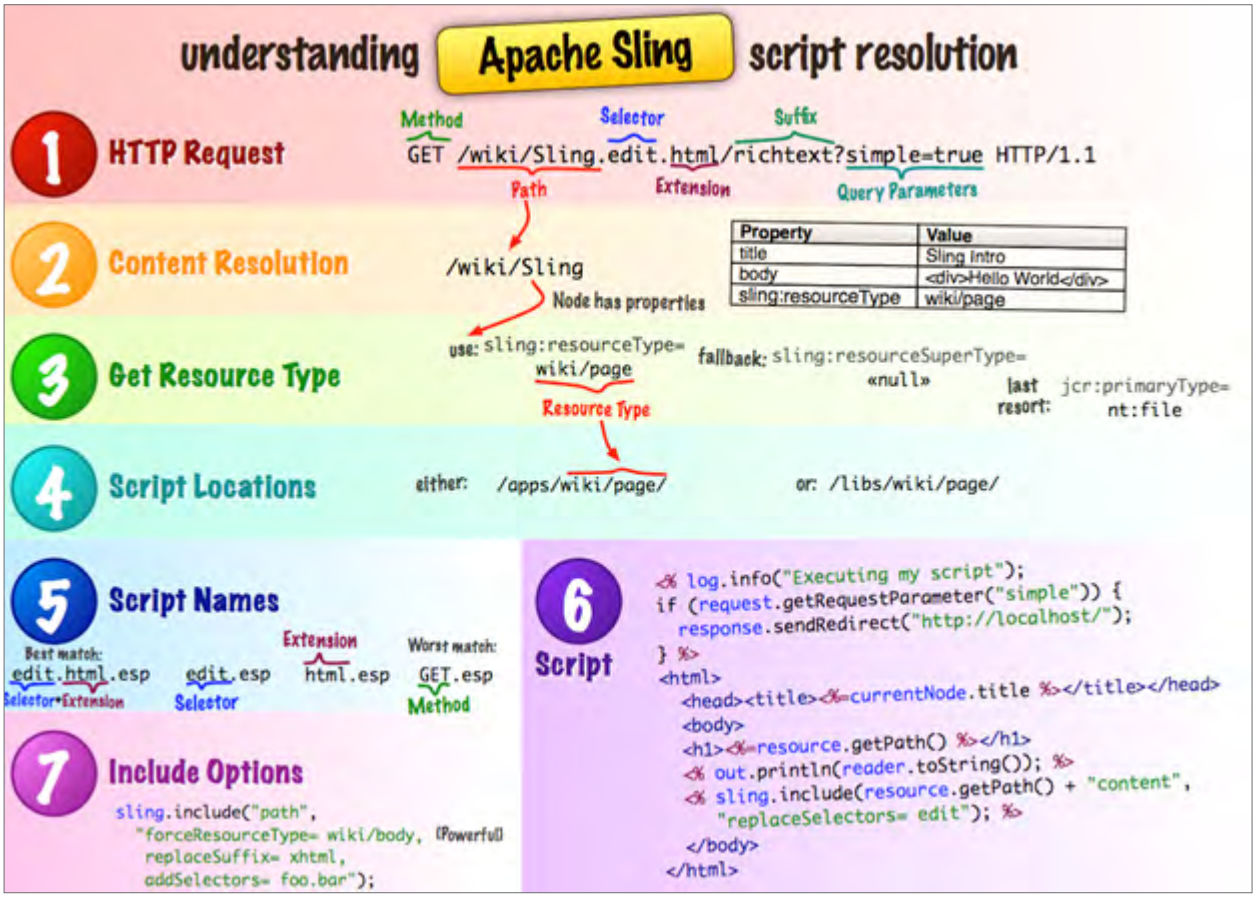
Locating and Rendering Scripts

When the resource is identified from the URL, its resource type property is located, and the value is extracted. This value is either an absolute or a relative path that points to the location of the script to be used for rendering the content. All scripts are stored in either the /apps or /libs folder and are searched in the same order. If no matching script is found in either of the folders, the default script is rendered. For multiple matches of the script, the script name with the best match is selected. The more selector matches, the better, as shown in the below screenshot:



URL Decomposition

Each content item in the JCR is exposed as an HTTP resource, and the request URL addresses the data to be processed. After the content is determined, the script or the servlet to be used to handle the request is determined. In the chronological order, the Sling Resource Resolver first tries to resolve the URL to a servlet or redirect rule. If it does not exist or is not successful, the script resolution process described in the below screenshot takes place. If no node is found, a 404 error is returned.



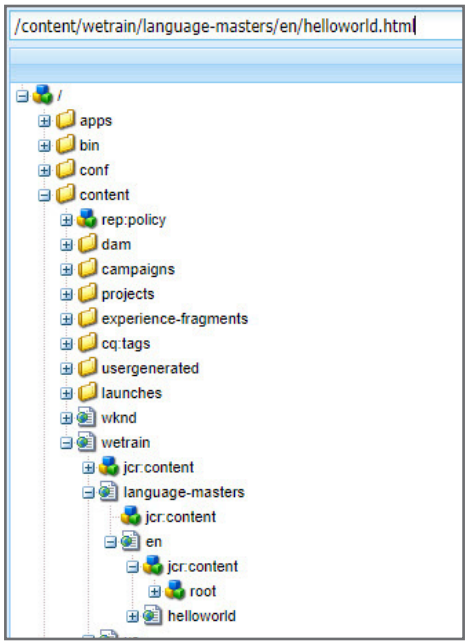
Exercise 3: Search for a Rendering Script

When presented with a request (URL), Sling performs the following actions:

1. Disregard the protocol, server, port, and any suffix from the URL.
2. Examine the remaining portion of the URL, use the information contained therein to assist with resource solution, and find the associated rendering script.

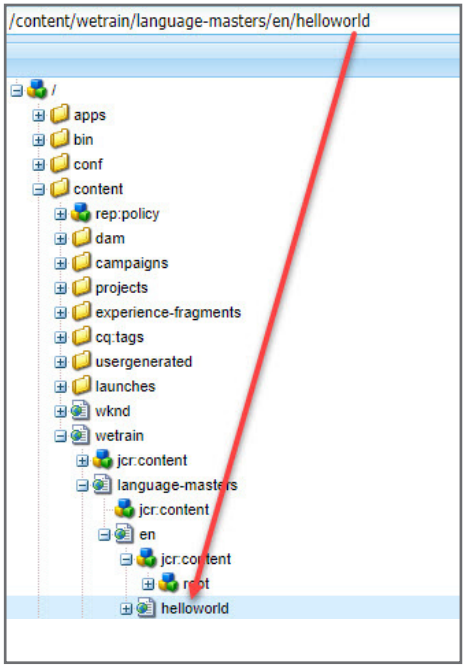
In this exercise, you will follow how Sling resolves a URL into resources and scripts to render. Now, let us follow Sling's actions to see how the we-train resource was rendered in the previous exercise.

1. Consider the request (URL) http://localhost:4502/content/wetrain/language-masters/en/helloworld.html
2. Disregard **http://localhost:4502**.
3. Ensure the **CRXDE Lite** page is open.
4. Navigate to **/content/wetrain/language-masters/en/helloworld.html**. Notice that the **.html** portion of the URL does not exist in the repository.



At this point, Sling backs off the remaining part of the request until the first period (.) and everything after the **.** is considered the extension.

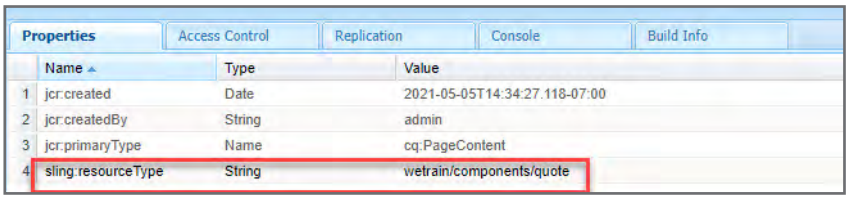
5. In **CRXDE Lite**, navigate to the **/content/wetrain/language-masters/en** node, as shown. You should find a node in the repository that matches the resource in the request. What you just did is called *resolving the resource*. You resolved a request URL to a resource that is represented by a node in the repository.



Note: You can map resources to the items in the relational databases and/or the file system. The Apache Sling specification does not mandate a JCR database.

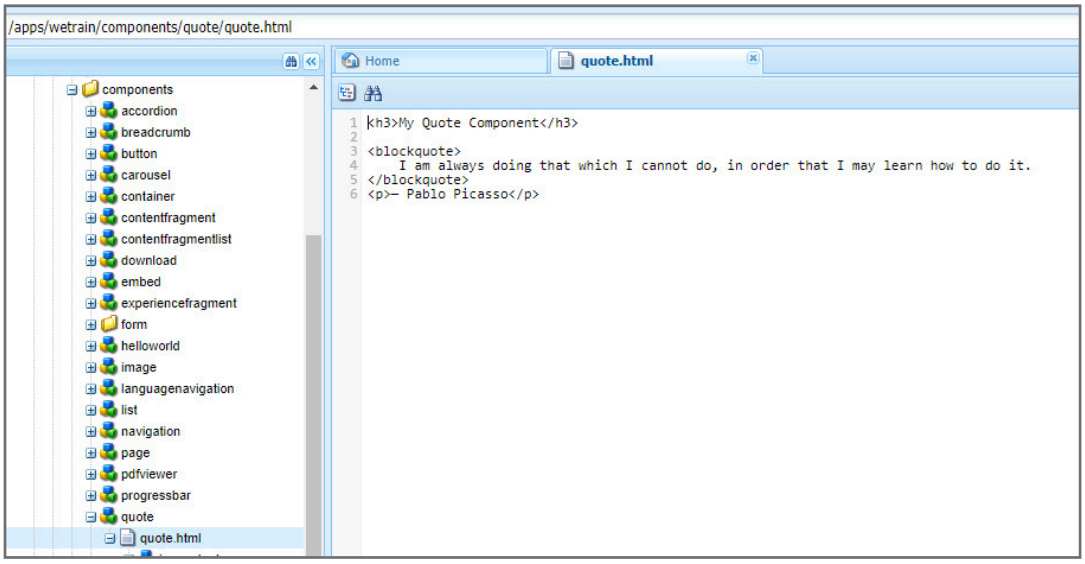
Now, you must find the rendering script.

6. Navigate to the **/content/wetrain/language-masters/en/helloworld** node and select the **sling:resourceType** property. The value of the **sling:resourceType property** is what Sling uses to begin its search for a rendering script. Notice that the **sling:resourceType** property points to the quote node you created earlier:

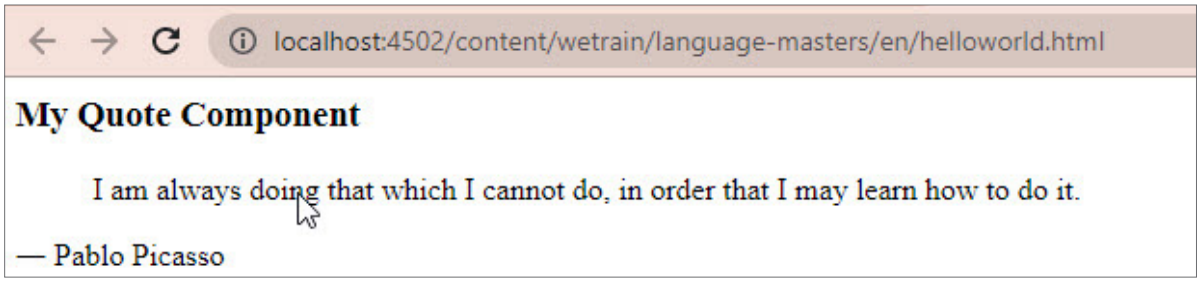


7. Navigate to the **/apps/wetrain/components/quote** node.

8. Notice the **quote.html** script. This is the default script because the script's name matches the name of the folder in which the script resides.



Later, you will explore many options that Sling might use in selecting the correct script. For this exercise, a default script is available. Given the specified request and the available selection of scripts, the default script is the best match and Sling chooses it to render the resource.
You have just seen how the request http://localhost:4502/content/wetrain/language-masters/en/helloworld.html results in the following browser output:



In this exercise, you have successfully resolved the resource, found the rendering script, and invoked the rendering chain.

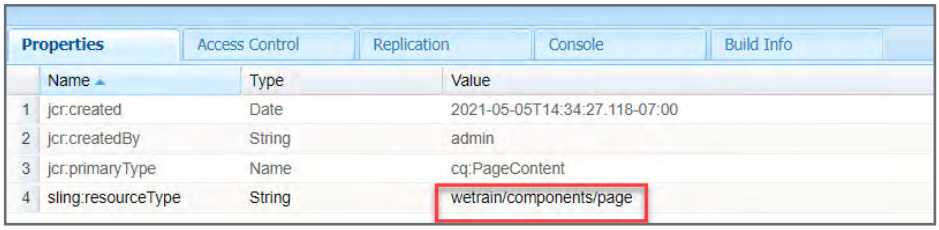
Exercise 4: Use the Authoring Interface to Add Content

To drag a component onto a page, the component needs to have a component group and a dialog box. The component group informs AEM whether the component can be added to different containers on the page. Content components are required to have a dialog, even if it is empty. In this exercise, you will create an empty dialog and a componentGroup to test the quote component on a page.

Task 1: Connect the helloworld Page to the Page Component

Now let us connect the page **helloworld** that we created in Exercise 2 to the page component. This will allow the other components to be dynamically rendered on to the page. To do this:

1. Navigate to **/content/wetrain/language-masters/en/helloworld/jcr:content.**
2. On the **Properties** tab, double-click **sling:resourceType** and change the value to **wetrain/components/page**.



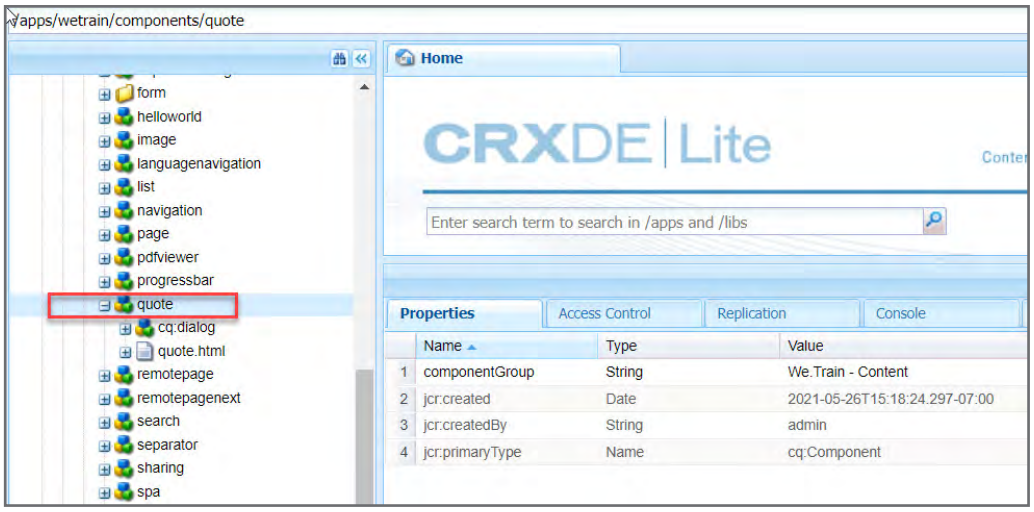
3. Click **Save All** to save the changes.
4. Select the **jcr:content** node and add the following property on the **Properties** tab:

Name	Type	Value
cq:template	String	/conf/wetrain/settings/wcm/templates/page-content

5. Click **Add**. The property is added to the node.
6. Click **Save All**.

Task 2: Use the Authoring Interface to Add Content

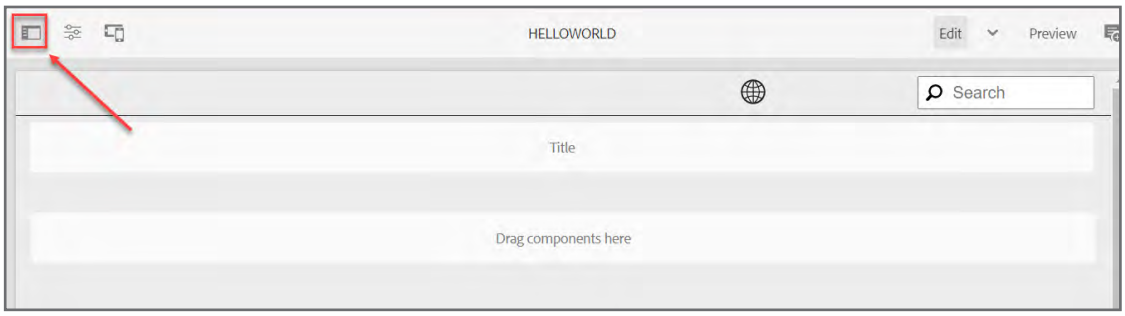
1. In **CRXDE Lite**, right-click the **quote** component and click **Create** > **Create Node**. The **Create Node** dialog box opens.
2. In the **Name** box, type **cq:dialog**.
3. From the **Type** dropdown menu, select **nt:unstructured** if it is not selected already.
4. Click **OK**. The node is created.
5. Click **Save All** to save the changes.
6. Add the following property to **/apps/wetrain/components/quote** component:
   a. **Name**: **componentGroup**
   b. **Type**: **String**
   c. **Value**: **We.Train - Content**
   d. Click **Add**. The property is added.
7. Click **Save All** to save the changes. The quote component properties should look like this:



8. In the AEM author instance, navigate to **Sites** > **We.Train** > **Language Masters** > **en** > **helloworld**, as shown:

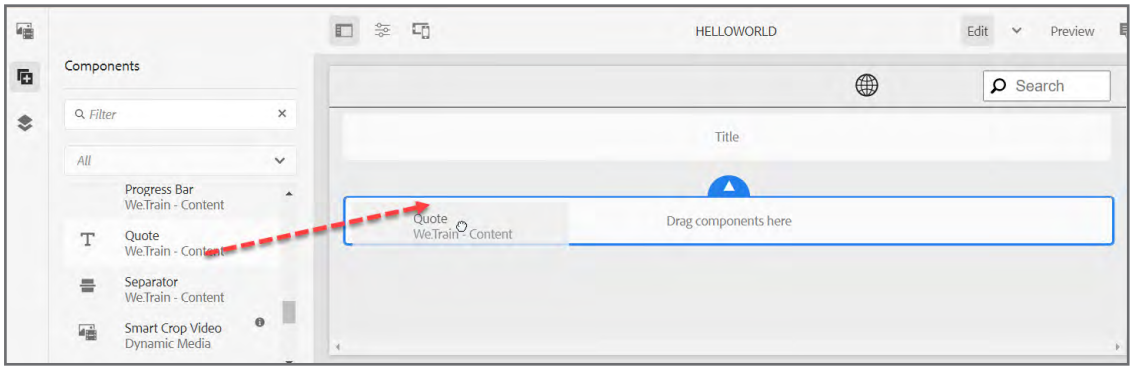
9. With the **helloworld** page selected, click **Edit (*e*)** from the top menu bar to open the **helloworld** page.

10. In the upper left, click the **Toggle Side Panel** icon, as shown, if you do not see the left panel.

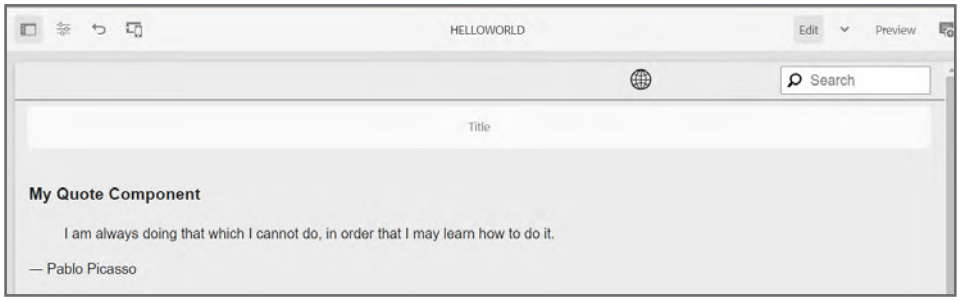


11. In the left panel, click the **Components** icon, as shown. A list of available components is displayed.

12. Look for the **Quote** component and drag and drop the component, as shown, onto the layout container:



Your page should look similar to the screenshot below:



Dialogs

Dialogs provide an interface for authors to configure and provide input to the component. Depending on the complexity of the component, the dialogs can have one or more tabs. The tab keeps the dialog short and sorts the input fields.

Coral UI and Granite UI

The Coral UI and the Granite UI define the look and functionality of AEM. The Granite UI provides a large range of the basic components (widgets) needed to create a dialog in the authoring environment. When necessary, you can extend this selection and create your own widget.

cq:dialog

The `cq:dialog (nt:unstructured)` node type is used to create a dialog.

The cq:dialog node:

- Defines the dialog for editing the content of the component.
- Is specific to the touch-enabled UI.
- Is defined by using Granite UI components.
- Has a property `sling:resourceType`, as the standard Sling content structure.
- Can have a property `helpPath` to define the context-sensitive help resource (absolute or relative path) that is accessed when the Help icon (the ? icon) is selected.
  - For out-of-the-box components, `helpPath` often references a page in the documentation.
  - If no helpPath is specified, the default URL (documentation overview page) is displayed.

Exercise 5: Create an Edit Dialog

In this exercise, you will create a dialog box that enables an author to add a value to a form field and then have that value display on the component.

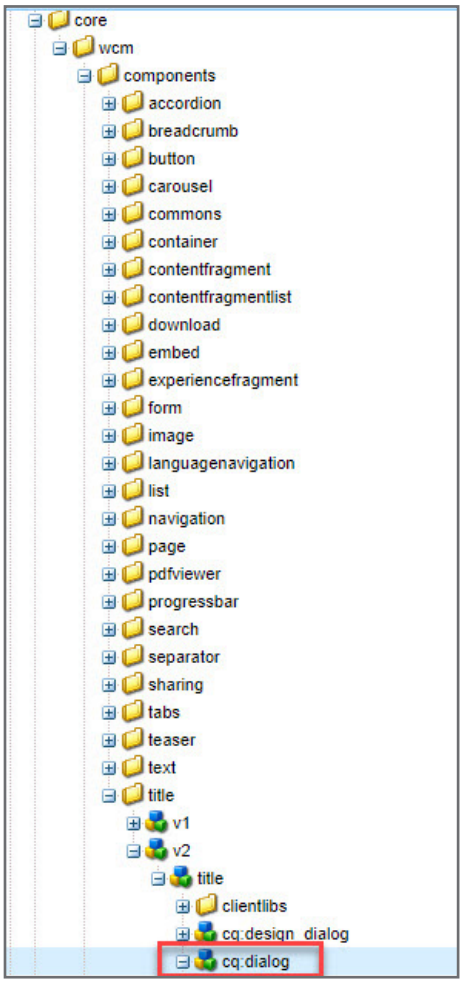
Task 1: Create an Edit Dialog

1. In **CRXDE Lite**, navigate to **/apps/wetrain/components/quote/cq:dialog**.
2. Right-click the **cq:dialog** node you created in the previous exercise and click **Delete**. The node is deleted. In this exercise, you will copy an existing dialog from a Core Component and modify it as per your requirement.
3. Click **Save All** to save the changes.

4. Navigate to **core/wcm/components/title/v2/title/cq:dialog:**

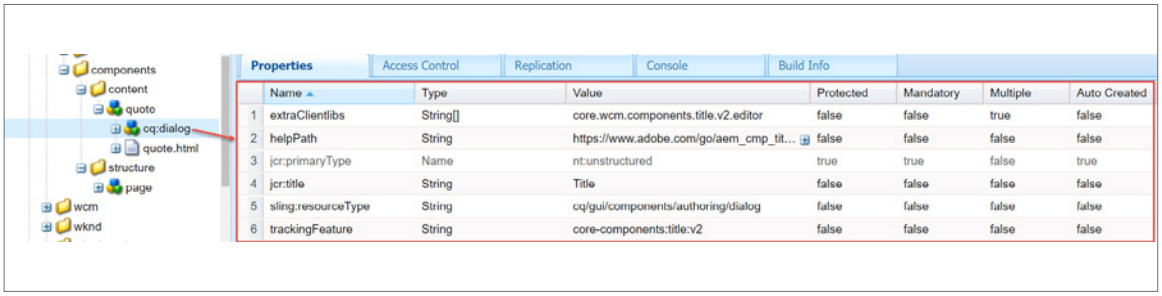
Note: Core Components are installed differently in 6.5 and Cloud Service. Use the locations below to find the Core Components:

- **6.5** use /apps/core...
- **Cloud Service** use /libs/core...



5. Right-click the **cq:dialog** node and click **Copy**. The node is copied.
6. Navigate to **/apps/wetrain/components/quote**.
7. Right-click the folder and click **Paste** to paste the node under the **quote** folder.

8. Select the **cq:dialog** node. You will see the properties of the **cq:dialog** node on the **Properties** tab on the right, as shown:



You must change the dialog title as this is a different component.

9. On the **Properties** tab, double-click **jcr:title** and change the value to **Quote**.

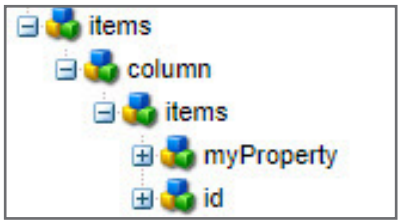
10. Under **quote/cq:dialog/content/items/tabs/items/properties/items/columns/items/column/items**:

    a. Right-click the **types** node and click **Delete** to delete the node.

    b. Right-click the **defaulttypes** node and click **Delete** to delete the node.

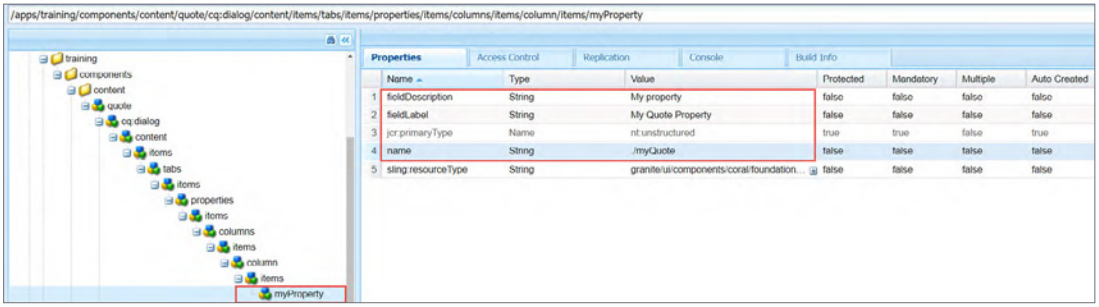
    c. Right-click the **linkURL** node and click **Delete** to delete the node.

    d. Right-click the **title** node and select **Rename** from the menu. Enter **myProperty** as the new node name.



    e. Click **Save All** to save the changes.

11. Select the **myProperty** node to update its properties:
    a. **fieldLabel**: **My Quote Property**
    b. **name**: **./myQuote**
    c. **fieldDescription:** Provide a suitable description.
    d. Click **Save All** to save the changes. The Properties tab should look, as shown here:



When an author uses this dialog and adds a value to the new formfield, AEM saves it under the current node on a property called myQuote. This value can then be accessed by the HTL script in the quote component.

12. Double-click **quote.html**. The editor opens on the right. Delete its content.

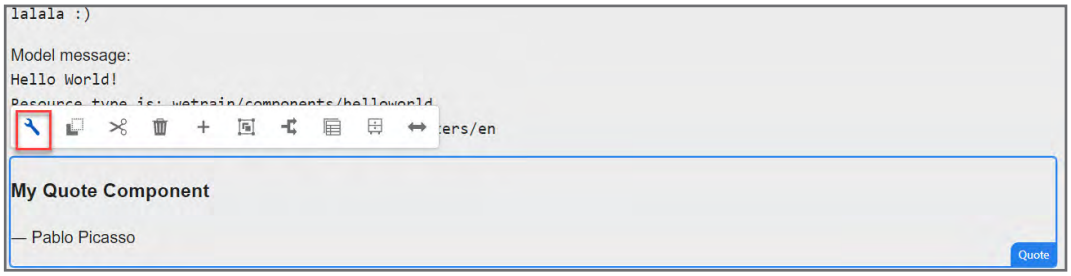
13. In the Exercise_Files-DWC folder provided to you by your instructor, navigate to the **ui.apps/src/main/content/jcr_root/apps/wetrain/components/quote** folder.

14. Open the **dialog-quote.html** file in Notepad++ (or any editor of your choice), copy the content of the file, and replace the existing content in the **quote.html** editor in **CRXDE Lite** with the new content.
    Notice the code `${properties.myQuote}`. By using the `${}` expression, you are able to insert an object into the HTML code. The properties object is a global object that gives access to all properties on the current resource.

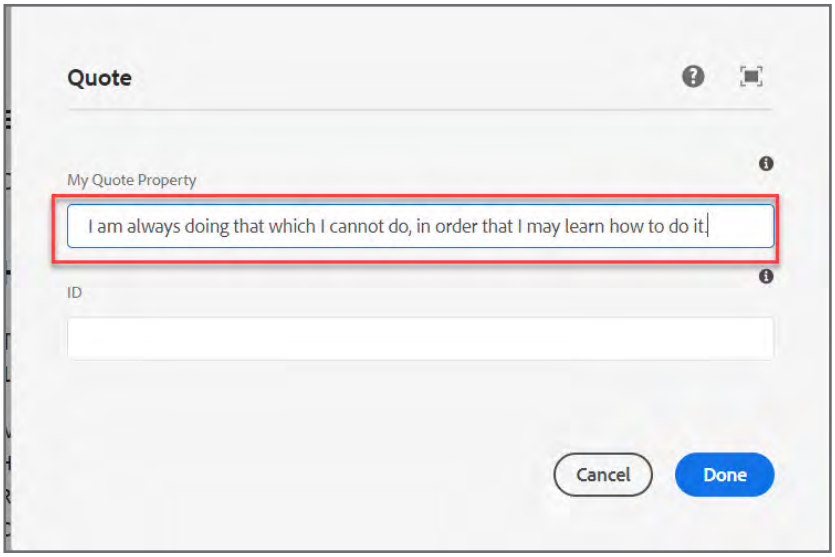
15. Click **Save All** to save the changes.

Task 2: Observe the Rendered Dialog Values

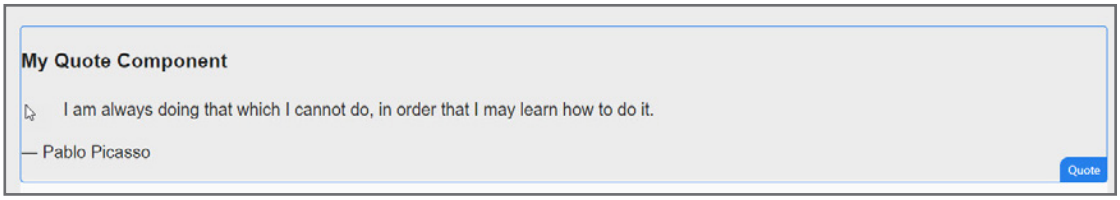
1. In the AEM author service, navigate to **Sites** > **We.Train** > **Language Masters** > **en** > **helloworld**.
2. With the **helloworld** page selected, click **Edit (*e*)** from the top menu bar to open the page on a new tab.
3. Click the **Quote** component on the page and click the **Configure** icon, as shown. The **Quote dialog box** opens. If your page does not have a Quote component, drag it onto the page.



4. Add the following value to the dialog box: **My Quote Property**: I am always doing that which I cannot do, in order that I may learn how to do it.



5. Click **Done**.
6. Observe the rendered dialog values on the page.



Exercise 6: Customize Selectors

A component can render content in several rendering variations. Each variation is described in its own script file. Selectors provide a way to choose the variation of the script that should be rendered.

Prerequisite: A Maven project imported into an IDE

Task 1: Customize Selectors

In this task, you will customize Sling to choose the script that you want to render.

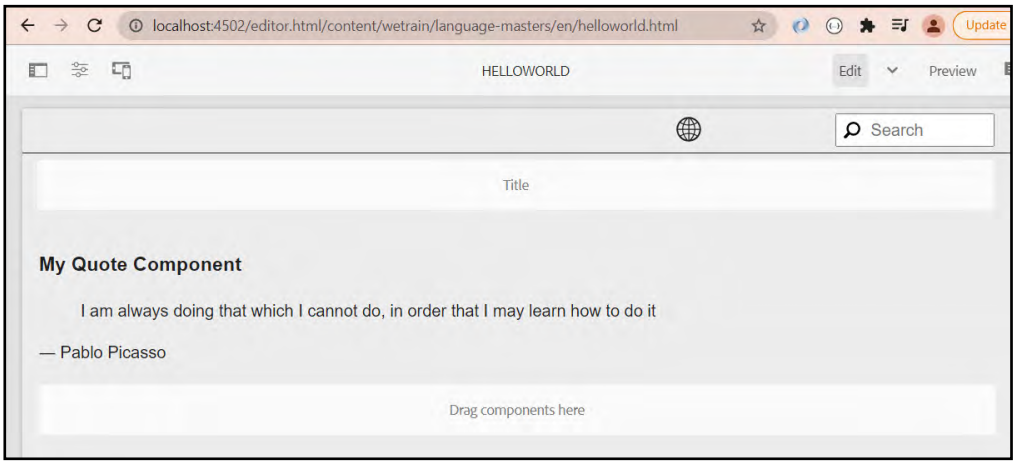
1. From **CRXDE Lite**, navigate to **/apps/wetrain/components/quote** node.
2. Select the **quote** node, right-click it and click **Create** > **Create File**. The **Create File** dialog box opens.
3. Enter **blue.html** in the **Name** box and click **OK**. The **blue.html** file is created.
4. Click **Save All**.

To add code to the **blue.html** page:

5. In the Exercise_Files-DWC folder provided to you, navigate to the **ui.apps/src/main/content/jcr_root/apps/wetrain/components/quote** folder.
6. Open the **blue.html** file in Notepad++ (or any editor of your choice), copy the content of the file and paste it in the **blue.html** editor in **CRXDE Lite**.
7. Click **Save All**.

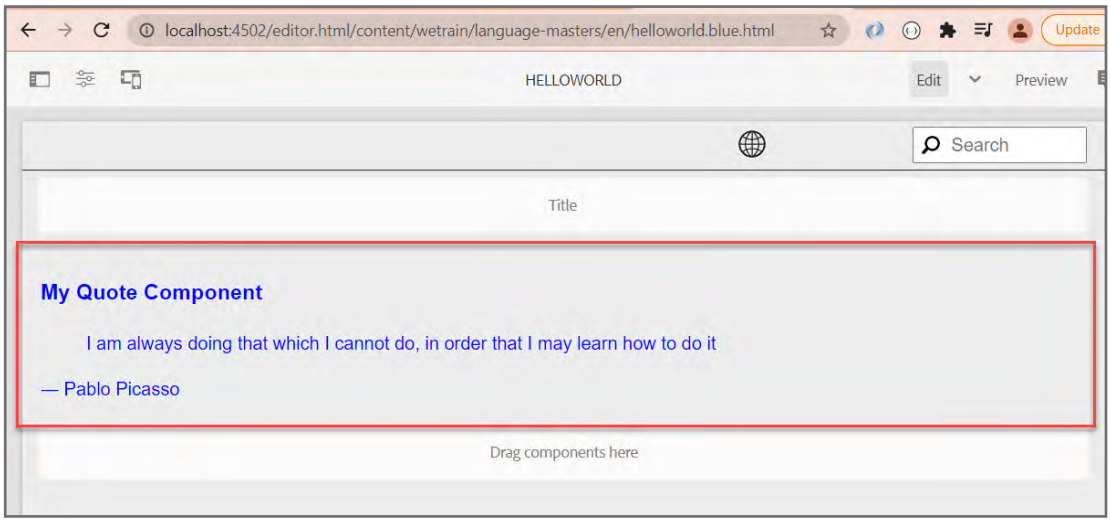
To test the selector:

8. Navigate to the wetrain helloworld page (http://localhost:4502/editor.html/content/wetrain/language-masters/en/helloworld.html).
   You added the Quote component to this page in one of the previous exercises. Notice that the text is in black font.



9. Add blue to the URL (http://localhost:4502/editor.html/content/wetrain/language-masters/en/helloworld.blue.html) and refresh the page.

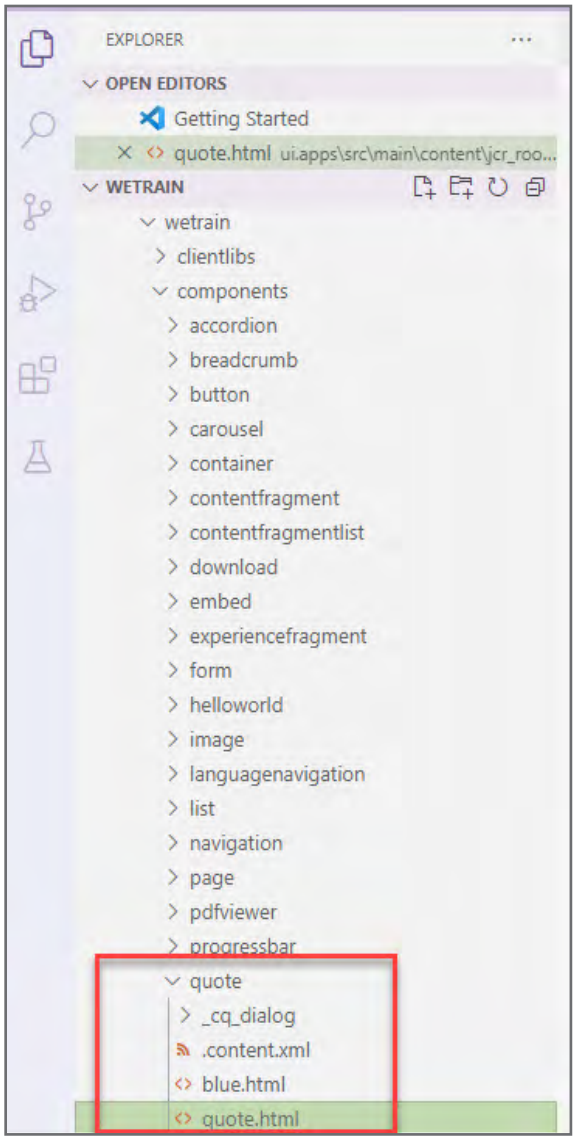
10. Notice the difference in font color. The code from **blue.html** is chosen as the script name that matches the selector in the URL, as shown:



Task 2: Import the Changes into the AEM Project

You made the quote component changes, directly in the AEM repository, using CRXDE Lite. You must now export the changes from the AEM repository and import them into your AEM project so that the changes will be retained when you next deploy your project.

1. In your IDE, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components.**
2. Right-click on the **components** component folder and select **Import from AEM Server**.
3. Expand the **quote** component folder. Verify the changes:



You have successfully imported the AEM changes to your IDE.

References

- Script Resolution
- URL Decomposition
- Sling Cheatsheet

Introduction to HTML Template Language

Introduction

HTML Template Language (HTL) is developed and supported by Adobe to replace Java Server Pages (JSP) in Adobe Experience Manager (AEM). HTL offers a highly productive enterprise-level web framework that provides increased security and helps HTML developers without Java knowledge to work on AEM projects easily.

Objectives

After completing this module, you will be able to:

- Explain HTL
- Explain the goals of HTL
- Explain the HTL syntax
- Render the page content by using AEM global objects
- Render page content by using HTL attributes

HTL

HTL is the recommended language for developing components in AEM. An HTL template defines an HTML output stream by specifying the presentation logic and the values that need to be inserted into the stream dynamically based on the background business logic.

HTL differs from other templating systems in the following ways:

- HTL is HTML5: A template created in HTL is a valid HTML5 file. All HTL-specific syntax is expressed within a data attribute or within HTML text. Any HTL file opened as HTML in an editor will automatically benefit from the features such as auto-completion and syntax highlighting that are provided by an editor for regular HTML.
- Separation of concerns (web designer versus web developer): The expressiveness of the HTL markup language is purposely limited. Only simple presentation logic can be embedded in the actual markup. All complex business logic must be placed in an external helper class. The HTL's Use API defines the structure of the external helper.
- Secure by default: HTL automatically filters and escapes all text being output to the presentation layer to prevent cross-site-scripting vulnerabilities.
- Compatibility: Compatible with JSP or ECMAScript Pages (ESP).

HTL: Goals

The main goals of HTL are to:

- Provide increased security through automated and context-sensitive cross-site scripting protection.
- Simplify development by helping HTML developers to write intuitive and efficient code.
- Reduce cost through reduced effort, faster Time To Market (TTM), and lower Total Cost of Ownership (TCO).

HTL Syntax

HTL uses an expression language to insert pieces of content into the rendered markup and HTML5 data attributes to define statements over blocks of markup such as conditions or iterations. As HTL is compiled into Java Servlets, the expressions and the HTL data attributes are both evaluated entirely server-side, and nothing remains visible in the resulting HTML.

Blocks and Expressions

HTL uses the following types of syntaxes:

- **Block Statements**: To define structural elements within the template, HTL employs the HTML data attribute. The data attribute is HTML5 attribute syntax intended for custom use by third-party applications. All HTL-specific attributes are prefixed with `data-sly-`.
- **Expression Language**: HTL expressions are delimited by characters ${ and }. At runtime, these expressions are evaluated, and their value is injected into the outgoing HTML stream. They can occur within the HTML text nodes or within the attribute values. In other words, you can include HTL expressions within HTML tags.

A list of a few basic HTL statements, expressions, and tags are:

- Comments. HTL comments are HTML comments with additional syntax. They are delimited as shown below:
  - Example: `<!--/* An HTL Comment */-->`
- Expressions:
  - Examples: `${true}, ${properties.text}`
- URI Manipulation:
  - Example: `${'example.com/path/page.html' @ scheme='http'}` (Resulting output: `http://example.com/path/page.html`)
- Enumerable objects:
  - Examples: pageProperties, properties, inheritedPageProperties
- HTL Block Statements:
  - use: `<div data-sly-use.nav="navigation.js">${nav.foo}</div>`
  - list: `<dl data-sly-list="${currentPage.listChildren}">`
  - `data-sly-include` and `data-sly-repeat`

- Special HTML tags:
  - Example: `<sly>`
- Expressions contain literals and variables.
  - Literals can be:
    - Boolean: `${true} ${false}`
    - Strings: `${'foo'} ${"answer"}`
    - Positive integers: ${42}
    - Arrays: `${[42, true, 'Hello World']}`
  - Variables are accessed through: `${properties.myVar}`

Exercise 1: Render Page Content by Using AEM Global Objects

Now that you have a basic understanding of component structure from the previous module, this module will teach you how to interact with those components within your Maven project and install them into your local quickstart. This allows you to use the full development features of your IDE while still testing in AEM.

Prerequisites:

- A Maven project imported into your IDE and installed on the author service
- The quote component from the previous module

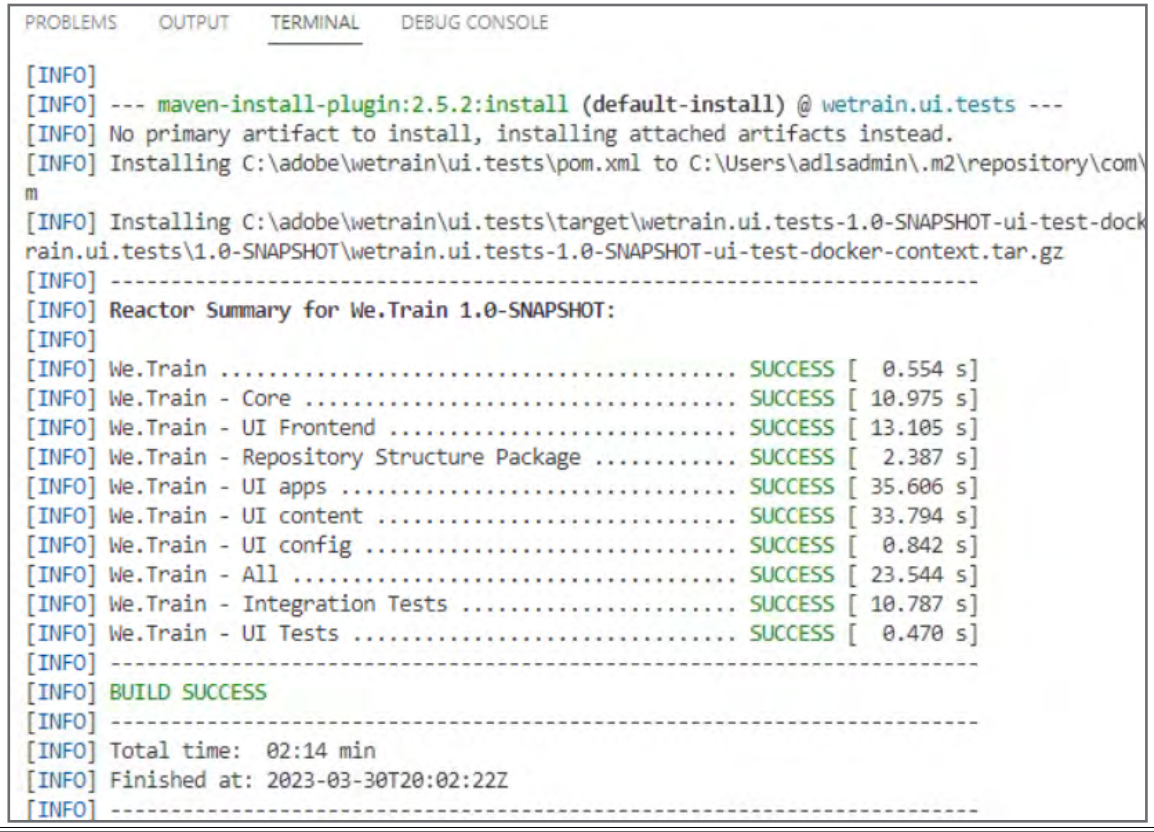
Task 1: Render Page Content

1. In your IDE, navigate to the **/apps/wetrain/components/quote** component.
2. Double-click the **quote.html** file. The editor opens on the right. Delete the existing content.
3. In the Exercise_Files-DWC folder provided to you, navigate to the **ui.apps** > **src/main/content** > **jcr_root/apps** > **wetrain/components/quote** folder.
4. Open the **globalobjects-quote.html** file in Notepad++ (or any editor of your choice), copy the content of the file, and replace the existing content in the **quote.html** editor in IDE with the new content.
5. Click **File** > **Save** to save the changes.

Task 2: Deploy and Verify

1. Using your IDE, open a terminal window.
2. Type in the following command to deploy your project to AEM:

```
mvn clean install -PautoInstallSinglePackage
```



Note: IDEs that have an AEM sync tool typically have a setting for autosync on save. If this setting is currently enabled, the updates you just performed have already imported into AEM and there is no need to run Maven.

3. In the AEM author service, navigate to **Sites** > **We.Train** > **Language Masters** > **en** > **helloworld**.

4. With the **helloworld** page selected, click **Edit** (***e***) on the actions bar to open the page in edit mode.

5. Examine the Quote component on the page. If you do not see the Quote component on the English page, drag the **Quote** component from the **Components** tab on the left panel on the page. Notice how different global objects are used for different content within AEM. Depending on what you are trying to accomplish, different objects can be used for desired values.

Exercise 2: Render Page Content by Using HTL Attributes

In this exercise, you will add a new script to the quote component so you can observe different HTL attributes and what the attributes are accomplishing.

Prerequisites:

- A Maven project imported into your IDE and installed on the author service
- The quote component from the previous module

Task 1: Render Page Content

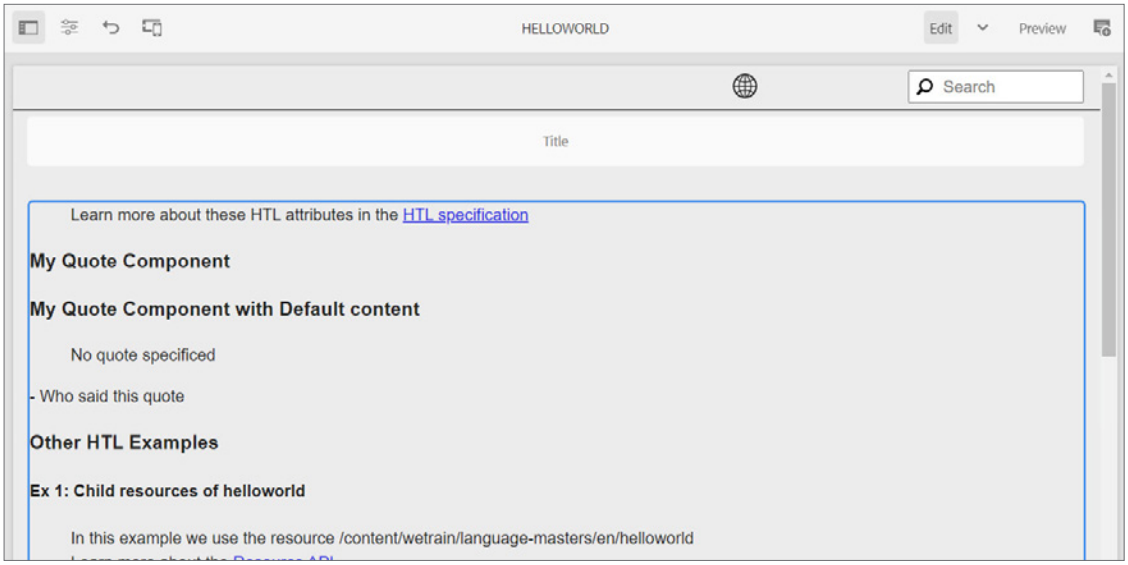
1. In your IDE, navigate to **/apps/wetrain/components/quote** component.
2. Double-click **quote.html** to update the code. The editor opens on the right. Delete its content.
3. In the Exercise_Files-DWC folder provided to you, navigate to the **ui.apps/src/main/content/jcr_root/apps/wetrain/components/quote** folder.
4. Open the **htl-quote.html** file in Notepad++ (or any editor of your choice), copy the content of the file, and replace the existing content in the **quote.html** editor in **IDE** with the new content.
5. Save the changes.

Task 2: Deploy and Verify

1. Using your IDE, open a terminal window.
2. Type in the following command to deploy your project to AEM:

```
mvn clean install -PautoInstallSinglePackage
```

3. In the AEM author service, navigate to **Sites** > **We.Train** > **Language Masters** > **en** > **helloworld**.
4. Open the **helloworld** page in edit mode and observe the newly rendered quote script. Notice how the different HTL attributes allow differently rendered HTML. Carefully inspect the comments in your new **quote.html** script to better understand what these attributes are accomplishing.



References

Experience Manager HTL Help

HTL Specifications:
https://github.com/Adobe-Marketing-Cloud/htl-spec
https://github.com/Adobe-Marketing-Cloud/htl-spec/blob/master/SPECIFICATION.md

Getting Started with Sites Components

Introduction

Components are the basic structural elements for Experience Manager. They render the content of the pages being authored. Components make page creation simple but powerful for the author and the development of components flexible and extensible for the developer.

The Core Components are a set of open-source, standardized Web Content Management (WCM) components for Adobe Experience Manager (AEM) to speed up development time and reduce maintenance cost of your websites.

Objectives

After completing this module, you will be able to:

- Describe components
- Describe Core components
- Investigate components
- Create proxy components
- Identify a proxy component and know when to use proxy components
- Identify AMP support provided by the Core components
- Identify Style System support provided by the Core components

Components

Components are the structural elements that constitute the content of pages. Modular and reusable, components are a fundamental element of Experience Manager. By using components constructing visually arresting and informative page is a simple process. In addition, the use of components to process and render content allows for flexible and extensible application implementation for the developer.

Developed using HTL (recommended) or JSP, components are essentially collection of scripts that implement a specific functionality and correspond to a Resource Type. Experience Manager components have a standardized user interface, work anywhere within an Experience Manager system and have no hidden configuration files. Experience Manager components use configuration dialogs to take author input and configure rendering behavior.

Components defined as container components can contain other components.

Core Components Overview

Open Source, extensible, cloud-ready and production ready, the Core components are a set of standardized WCM components for Experience Manager to speed up development time and reduce maintenance cost of your websites. They provide a robust and extensible set of base components for Experience Manager, built using the latest technology and best practices.

The Core components are versioned so you will not be surprised by new behaviors when new releases are applied. This versioning feature makes it easier to choose when you want to use the capabilities of newer version of the components.

Core components are headless/SPA-ready out of the box. Streamlined JSON output allows client-side rendering, still with the capability of in-context editing.

The Core Component Library provides an excellent reference for exploring and learning the capabilities and behaviors of each Core component.

Note: You will find the Core components at **/libs/core/wcm/components** for Cloud Service and at **/apps/core/wcm/components** for 6.x.

Proxy Components

Content resources have a `sling:resourceType` property that references the component responsible for rendering that content. It is good practice to have these properties pointing to site-specific components. As a result, Core components must not be referenced directly on the page. Instead, project- or site-specific components should be created. Project- and site-specific components offer more flexibility and avoid content refactoring if one site needs a different behavior for the same component.

However, for the project- or site-specific components not to duplicate any code, these components, which define the desired component name and component group to display to page authors might refer to Core Components as their super-types using the **sling:resourceSuperType** property. Project- and site-specific components that refer to a Core component parent and don't have much customization are called proxy components. Proxy components can be entirely empty if they fully inherit the functionality, or they can redefine some aspects of the component.

Exercise 1: Investigate Components

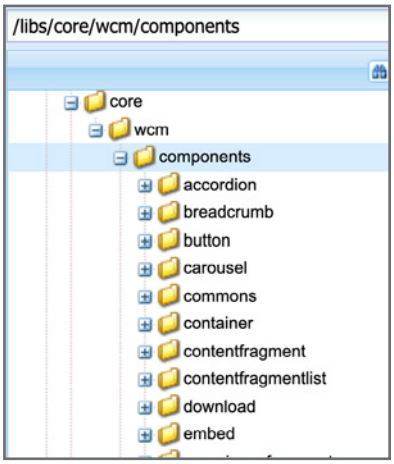
In this exercise you will investigate the Core Components and the wetrain proxy components.

Prerequisites:

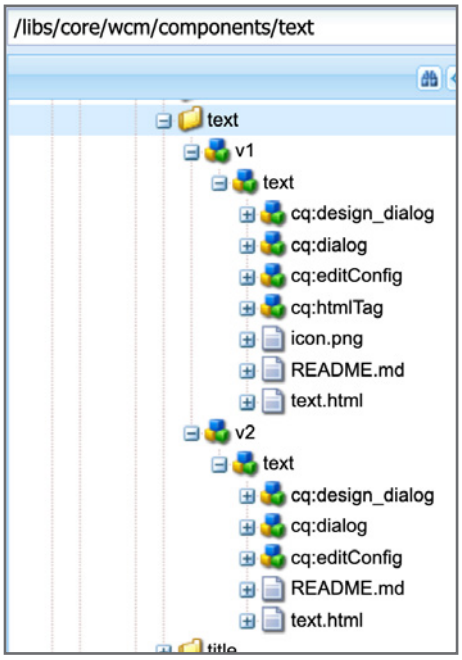
- Local author service running
- We.Train project deployed to AEM
- For AEM 6.5 only: Core Components are deployed to AEM

Task 1: Investigate the Core Components

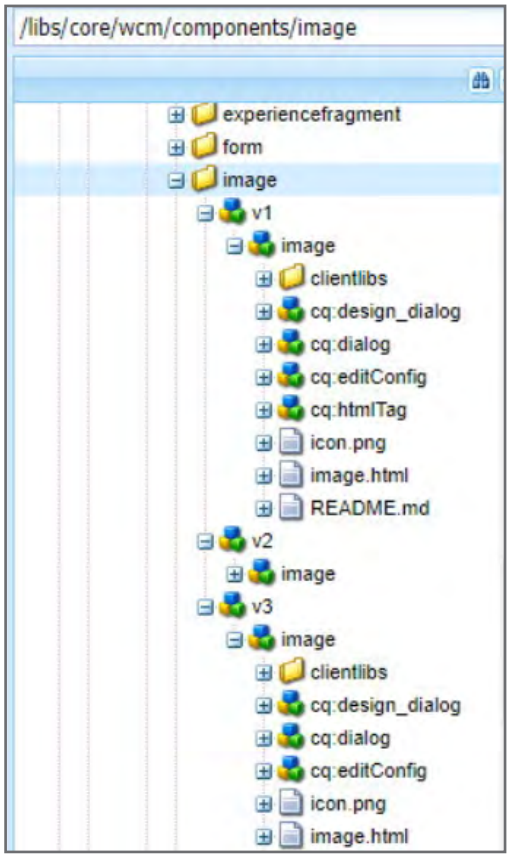
1. Using **CRXDE Lite**, navigate to **core/wcm/components**. For Cloud Service, the Core Components are 'shipped' with AEM and are located in **/libs**. For 6.x, you must install the Core Components, typically as part of the Maven build, and they are located in **/apps**.
   These are the Core Components.



2. Select and expand the **text** node. You will notice that there are two versions of the **text** component. Each version is a complete component, with scripts, dialog, design dialog, and editConfig.



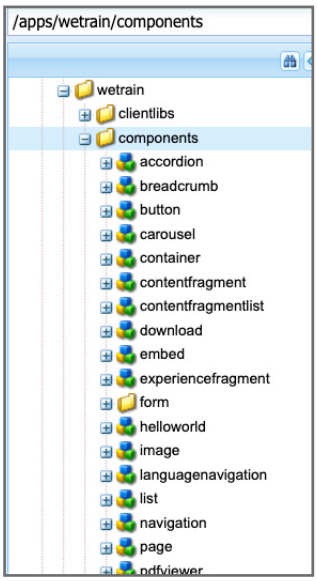
3. Select and expand the **image** node. You will notice that there are three versions of the image component. Each version is a complete component, with scripts, dialog, design dialog, and editConfig. The Core Image component also has its own clientlibs to configure its look and feel.



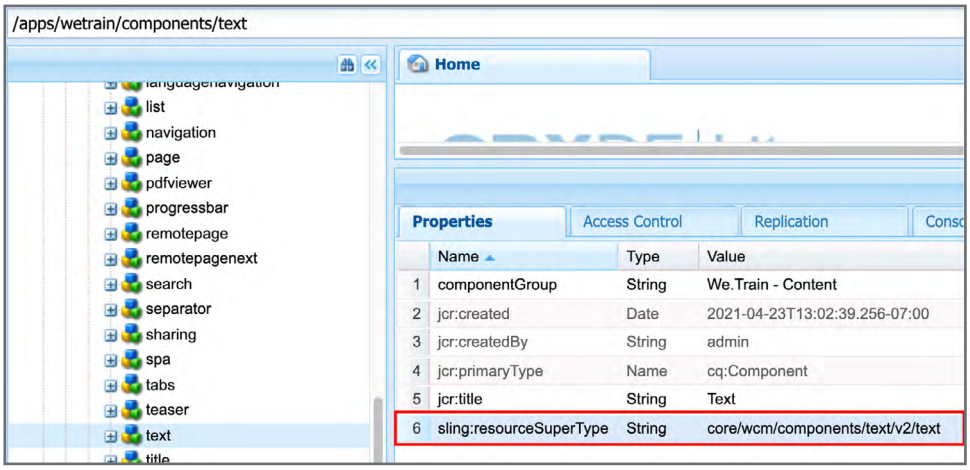
4. Take a few moments to explore and investigate the other Core Components in order to become familiar with their structure.

Task 2: Investigate the wetrain Proxy Components

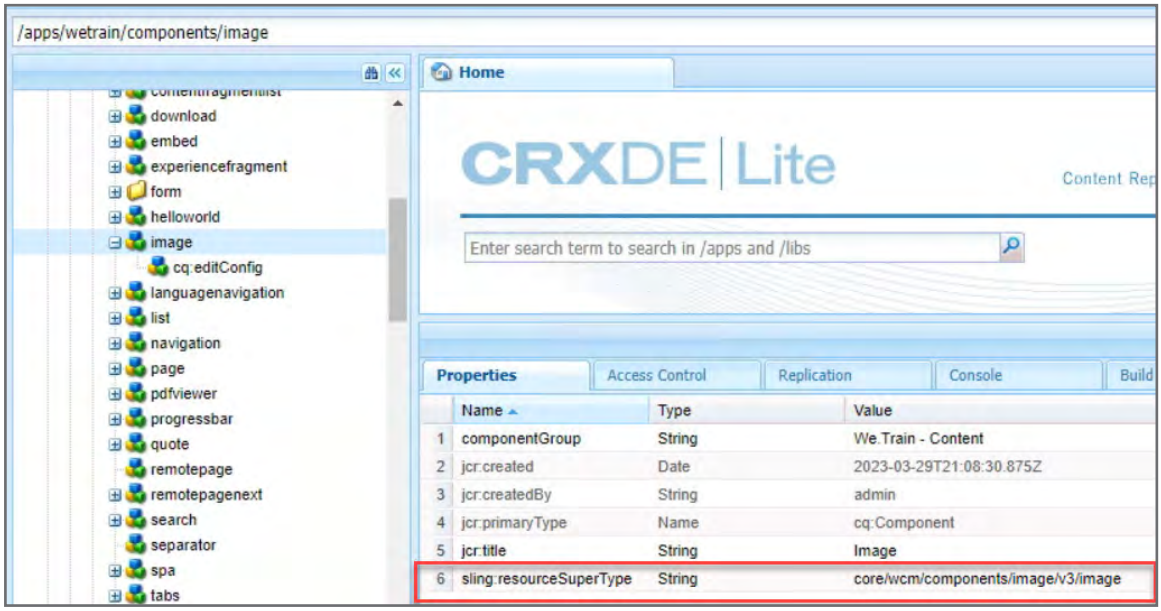
1. Using **CRXDE Lite**, navigate to **/apps/wetrain/components**, as shown:



2. Select the **text** component node. Notice the **slingResourceSuperType** property is set to **core/wcm/components/text/v2/text**.



3. Select the **image** component node and expand it. Notice the **slingResourceSuperType** is set to **[/apps | /libs]/core/wcm/components/image/v3/image**.



4. Take a few moments to explore and investigate the remaining **wetrain** proxy components that were generated from the AEM Project Archetype.

Exercise 2: Install Latest Core Components (Optional AEM 6.5.X only)

When you use Archetype 27 build script to create a project for AEM 6.5.X, it sets Core Components version to 2.15.2. At any time, you can update your AEM instance to include the latest Core Components. Though updating to the latest Core Components is optional for your project, you should typically plan to update the Core Component version routinely to ensure you have access to the latest functionality. In this exercise you will update Core Component version, then deploy and verify.

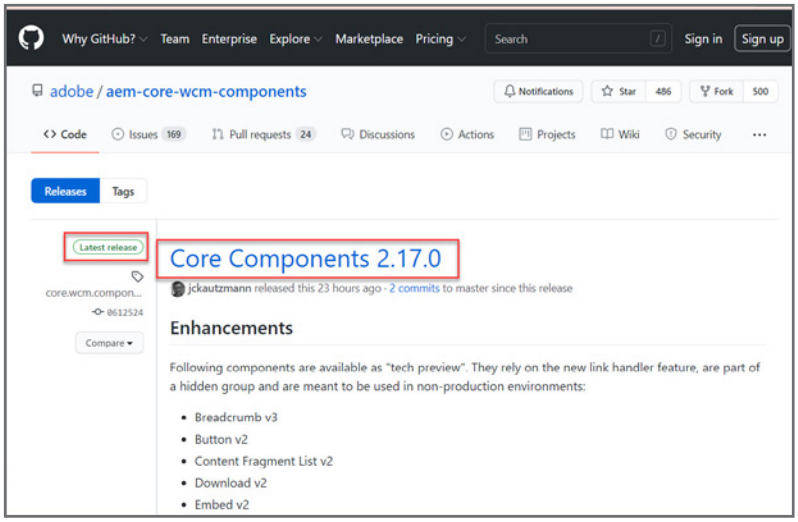
Prerequisites:

- Local author service running
- We.Train project deployed to AEM
- For AEM 6.5 only: Core Components are deployed to AEM

Task 1: Update Core Component Version

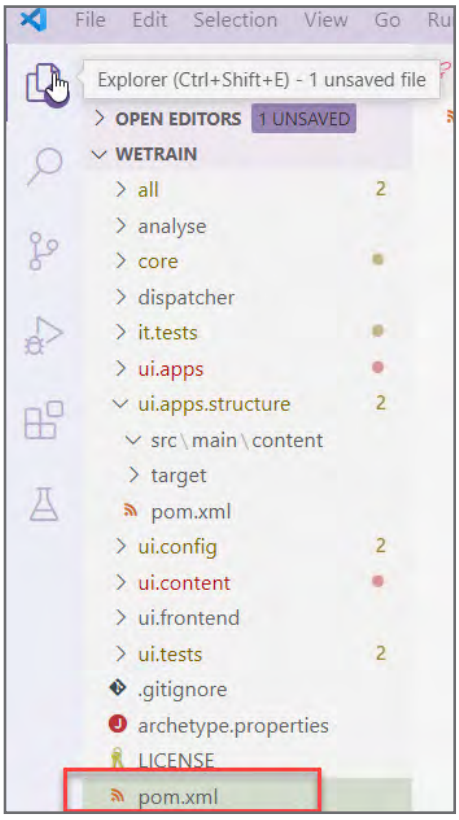
To find the latest version of Core Components:

1. Open https://github.com/adobe/aem-core-wcm-components/releases in a browser.
2. The aem-core-wcm-components Releases are displayed. You can see the **Latest release:**

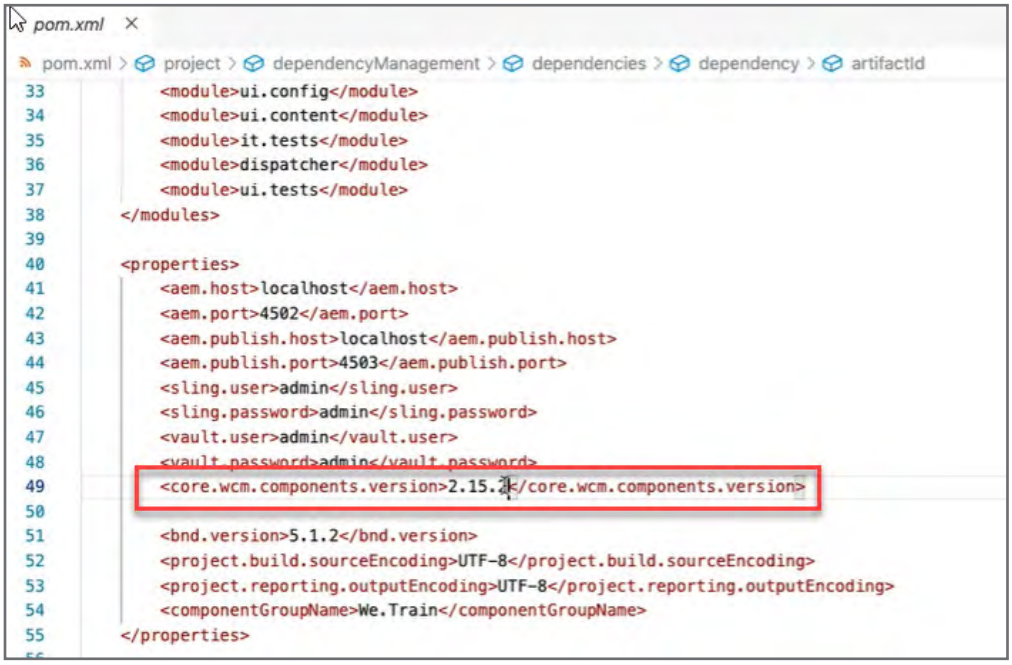


To update the version of Core Components for AEM 6.5 project:

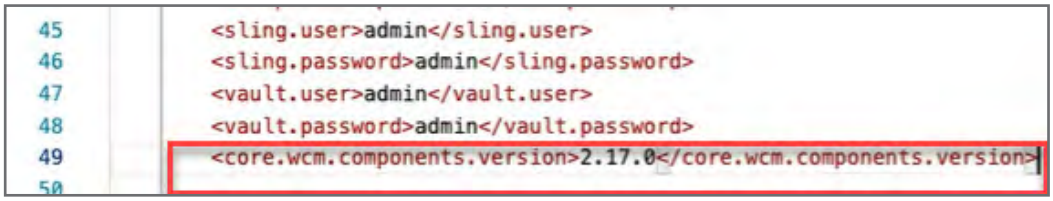
3. In your IDE, navigate to **WETRAIN** > **pom.xml**, as shown:



4. Open **pom.xml** in editor and verify the **core.wcm.components.version** around line number 49, as shown:



5. Update the **core.wcm.components.version** to **2.17.0,** as shown:



6. Save the changes (**File** > **Save**).

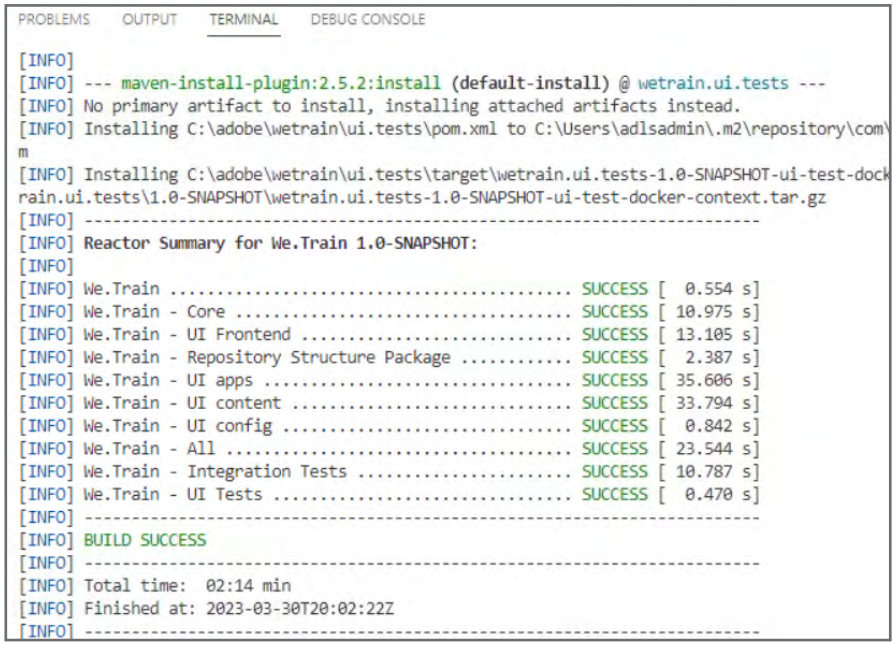
Task 2: Deploy and Verify

1. In your IDE, open a terminal window.

Note: An alternative to the IDE terminal window you could open an OS terminal window in your <AEM Project> folder.

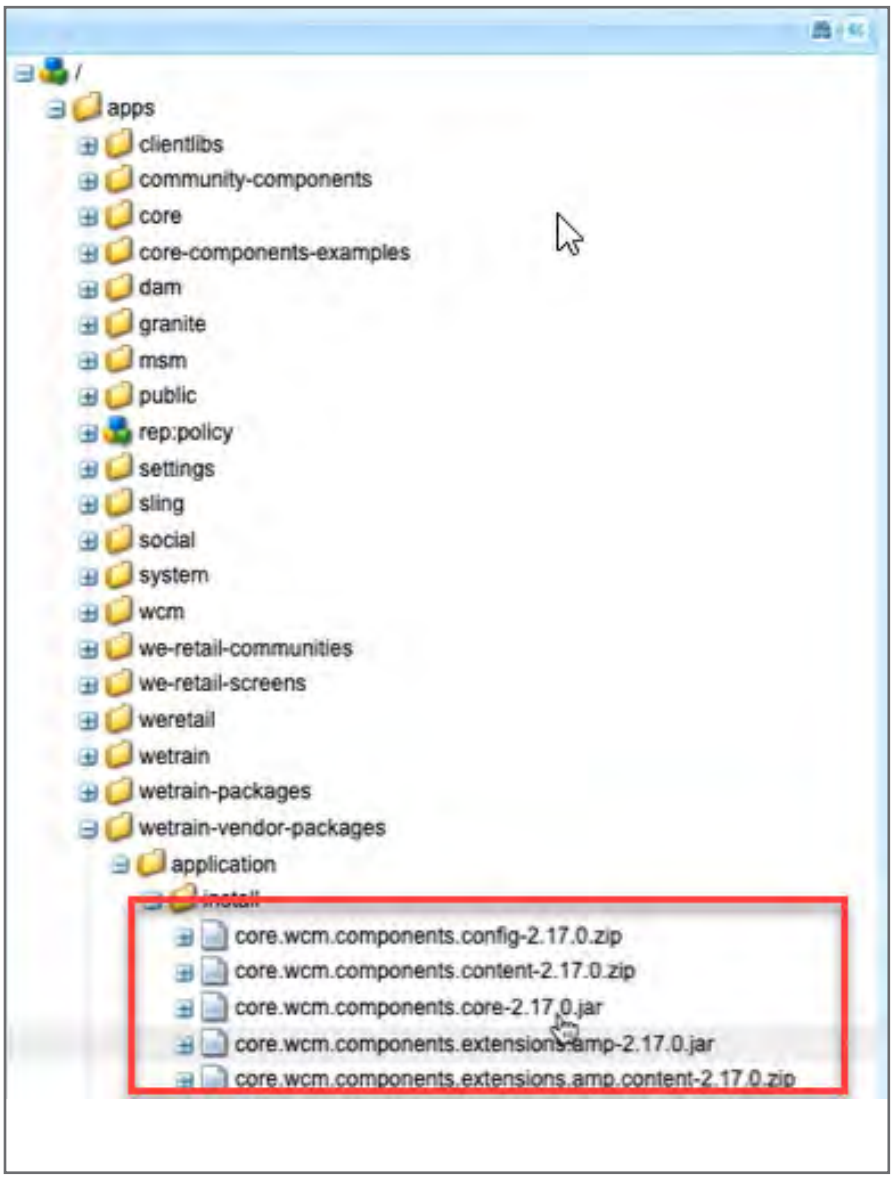
2. Type in the following command to deploy your project to the author service:

```
mvn clean install -PautoInstallSinglePackage
```



To verify the project deployment:

3. Use a browser and open **CRXDE Lite**.
4. Navigate to **apps/wetrain-vendor-packages/application/install,** as shown:



Notice the **core.wcm.components.version** is updated **2.17.0.**

Experience Manager Features Enabled by Core Components

The following topics in this section are an introduction to some of the features enabled by the Core Components. These topics will be covered in more detail later in the course.

Responsive Editing

The Container component is the content area into which the page author will add components. Each container on the page is configured to manage its own allowed components. The container component is configured with a grid system that provides breakpoints to allow resizing of both the components in the container and the container itself. You can use the responsive grid to specify where, if, and how components will display depending on view screen size.

Experience Manager recognizes the responsive layout on pages using a combination of:

- Container component provides a responsive grid to change component size and position
- Layout Mode to change the size and layout of components, hide and float components to a new line
- Emulator simulates different screen sizes

Responsive Grid

The responsive grid defines differing content layouts, based on device width by using breakpoints. The grid allows component placement in the grid.

- Resizes components as required
- Defines when components should collapse/reflow by using the horizontal 'snap to grid'

By using the responsive grid in cooperation with layout mode, you can specify if/when components should be hidden, perhaps for smaller screen sizes.

Component Styling

Core Components follow a standardized naming convention to make styling easy. For example, the CSS class name cmp-list--upnext is of the form

```
'cmp - <component name> -- <style name>'
```

where cmp stands for component

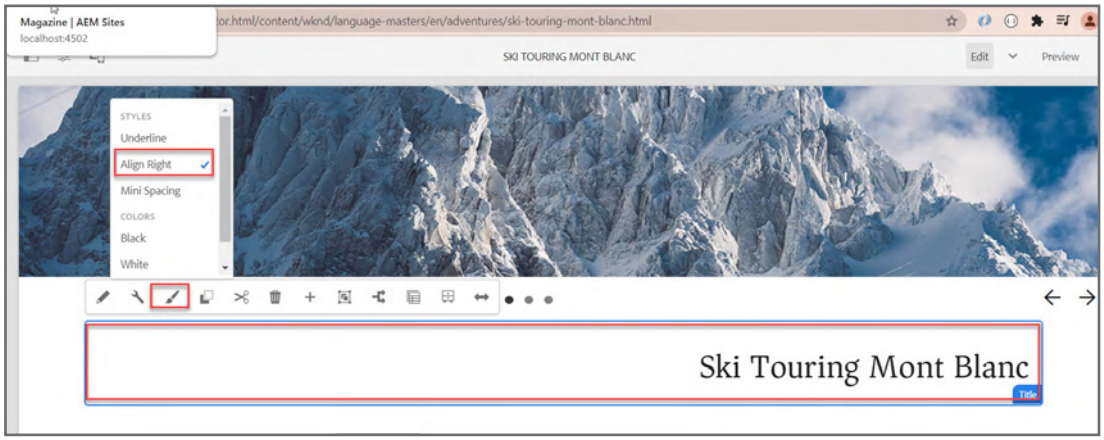
Each component is wrapped in a element with the cmp-<component-name> class. The following excerpt containing the wrapping element from the Core Text component HTL script illustrates this concept. Notice line 7:

```
1  <div data-sly-use.textModel="com.adobe.cq.wcm.core.components.models.Text"
2       data-sly-use.component="com.adobe.cq.wcm.core.components.models.Component"
3       data-sly-use.templates="core/wcm/components/commons/v1/templates.html"
4       data-sly-test.text="${textModel.text}"
5       data-cmp-data-layer="${textModel.data.json}"
6       id="${component.id}"
7       class="cmp-text">
8      <p class="cmp-text__paragraph"
9         data-sly-unwrap="${textModel.isRichText}">${text @ context = textModel.isRichText ? 'html' : 'text'}</p>
10 </div>
```

In line 8, you will notice that the CSS classes defined for the text component also use the cmp-<component name> format.

Style System Support

All Core Components implement the Style System. As a result, template authors can define additional CSS class names that can be applied to a component by page authors.



Dynamic Media Support

The Core Image component has built-in support for Dynamic Media. This built-in support allows a content author to use features of Dynamic Media when authoring pages:

- Image Presets
- Smart Crop
- Image Modifiers

The Dynamic Media HTTP commands support additional, advanced image modifications, for example:

- Color correction
- Basic templating and text rendering
- Localization
- Non-core Dynamic Media services: SVG, Image Rendering, and Web-to-Print

References

- Core Components Introduction:
  https://experienceleague.adobe.com/docs/experience-manager-core-components/using/introduction.html?lang=en#components
- Core Components GitHub:
  https://github.com/adobe/aem-core-wcm-components
- Component Library:
  https://www.aemcomponents.dev/
- Create Proxy Components:
  https://experienceleague.adobe.com/docs/experience-manager-core-components/using/get-started/using.html?lang=en#get-started
- Component Guidelines:
  https://experienceleague.adobe.com/docs/experience-manager-core-components/using/developing/guidelines.html?lang=en#developing
- Style System:
  https://experienceleague.adobe.com/docs/experience-manager-65/authoring/siteandpage/style-system.html?lang=en#siteandpage
- Core Component Dynamic Media:
  https://experienceleague.adobe.com/docs/experience-manager-learn/assets/dynamic-media/dynamic-media-core-components.html?lang=en#dynamic-media

Develop Designs for Components

Introduction

Modern websites rely heavily on client-side processing driven by complex JavaScript (JS) and Cascading Style Sheets (CSS) code. Organizing and optimizing the code can be a complicated task. Adobe Experience Manager (AEM) provides client-side library folders to store client-side code in the repository, organize them into categories, and define when and how each category of code can be served to the client. The client-side library system helps produce the correct links on the final webpage to load the correct code.

Objectives

After completing this module, you will be able to:

- Explain how the client-side libraries work
- Explain the client libraries' structure
- Organize client-side libraries
- Investigate a Client Library
- Investigate the ui.frontend Maven module
- Add content to a Page
- Start Webpack server for Front-end development
- Create a design for a component

Client-Side Libraries

The standard way to include a client-side library (a JS or a CSS file) in the HTML of a page is to include a <script> or <link> tag in the Java Server Page (JSP) containing the path to the JS/CSS file. Although this approach works, it can lead to problems when pages and their constituent components become complex. In such cases, multiple copies of the same JS library may be included in the final HTML output.

To avoid this issue, client-side library folders are used to organize the client-side libraries logically. The basic goals of client-side libraries are to:

- Store CSS/JS in small discrete files for easier development and maintenance
- Manage dependencies on third-party frameworks in an organized way
- Minimize the number of client-side requests by concatenating CSS/JS into one or two requests
- Minimize CSS/JS that is delivered to optimize speed and performance of a site

Node Structure

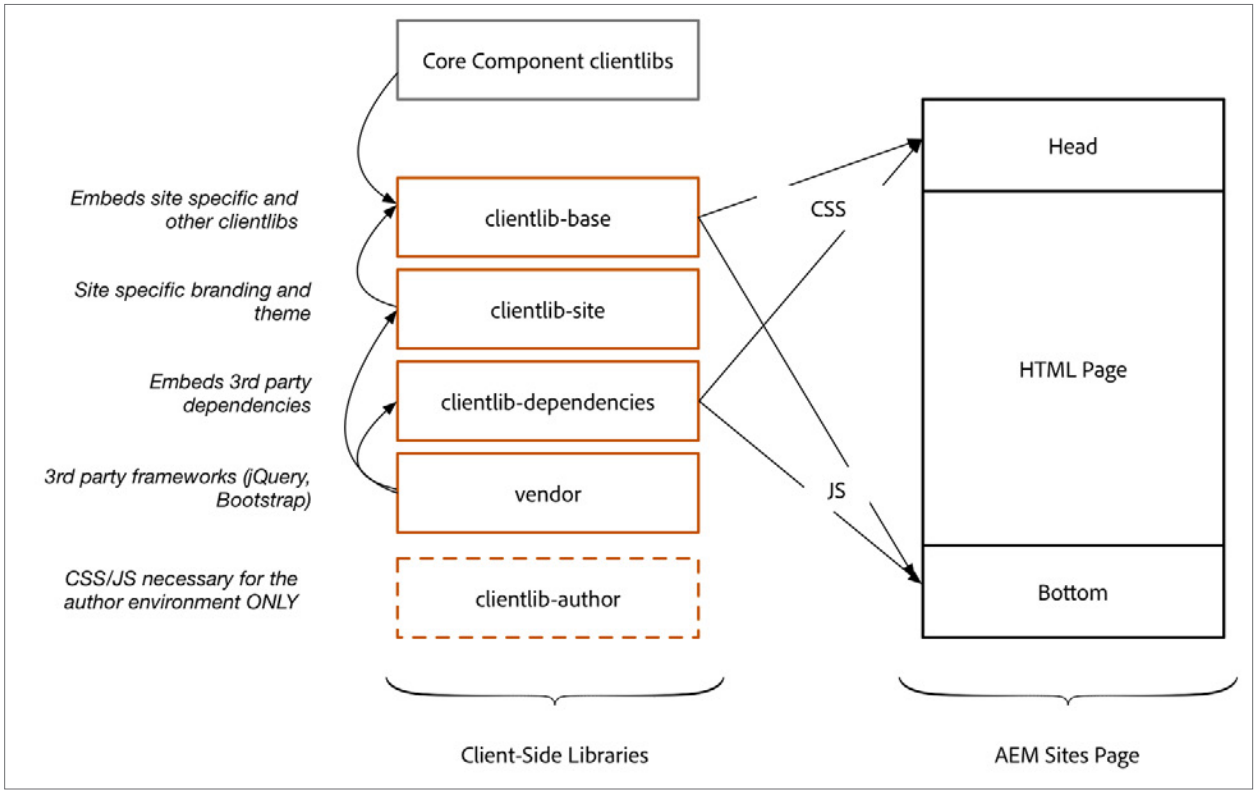
A client-side library folder is a repository node of type **cq:ClientLibraryFolder**. The typical node structure of a client library is:

```
[cq:ClientLibraryFolder] - jcr:primaryType
     - categories (String[])
     - embed (String[])
     - dependencies (String[])
     + css.txt (nt:file)
     + js.txt (nt:file)
```

Each cq:ClientLibraryFolder is populated with a set of JS and/or CSS files, along with a few supporting files. You can configure cq:ClientLibraryFolder by using the following properties:

- categories: This property helps identify the categories into which the set of JS and/or CSS files within the cq:ClientLibraryFolder belong. The categories property is multi-valued and enables a library folder to be a part of more than one category.
- dependencies: This property provides a list of other client library categories on which the library folder depends. For example, given two cq:ClientLibraryFolder nodes F and G, if a file in F requires another file in G to function properly, then at least one of the categories of G should be among the dependencies of F.
- embed: This property is used to embed code from other libraries. If node F embeds nodes G and H, the resulting HTML will be a concatenation of content from nodes G and H.
- allowProxy: If a client library is located under /apps, the allowProxy property allows access to it through the proxy servlet.
- js.txt/css.txt: This file helps identify the source files to merge in the generated JS and/or CSS files.

The below diagram explains the client-side libraries' convention followed in the We.Retail site (that is available out-of-the-box), and this convention can be applied to most sites' implementations:



Organizing Client-Side Libraries

By default, the cq:ClientLibraryFolder nodes can exist anywhere within the /apps and /libs subtrees of the repository. The common locations for clientlibs are:

- Site-level (/apps/<your-project>/clientlibs)
- Component-level (/apps/<your-project>/components)

Site-Level

The clientlibs at the site-level:

- Are used for CSS/JS for site specific design elements
- Have CSS at the top of a page and JS at the bottom of a page
    - › Examples include responsive design, dependencies, embedding structure components, and vendor code

Component-Level

The clientlibs at the component-level:

- Are component-specific CSS/JS
- Can be embedded into the site design

Referencing Client-Side Libraries

You can include the client-side libraries in HTML Template Language (HTL), which is the preferred technology. However, you can also use JSP.

In HTL, client-side libraries are loaded through a helper template. You can access the templates through `data-sly-use`. The three templates, CSS, JS, and all can be called through `data-sly-call`:

- CSS: Loads only the CSS files of the referenced client libraries.
- JS: Loads only the JavaScript files of the referenced client libraries.
- all: Loads all files of the referenced client libraries (both CSS and JavaScript).

Each helper template expects a categories option for referencing the desired client libraries. This option can be either an array of string values or a string containing a comma-separated values (CSV) list.

Exercise 1: Investigate a Client Library

Modern websites rely heavily on client-side processing driven by complex JavaScript and CSS code. Organizing and optimizing the serving of this code can be a complicated issue.

In this exercise, you will investigate a base client-side library. This base library will help compile the site's CSS and JS into a single client library.

Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service

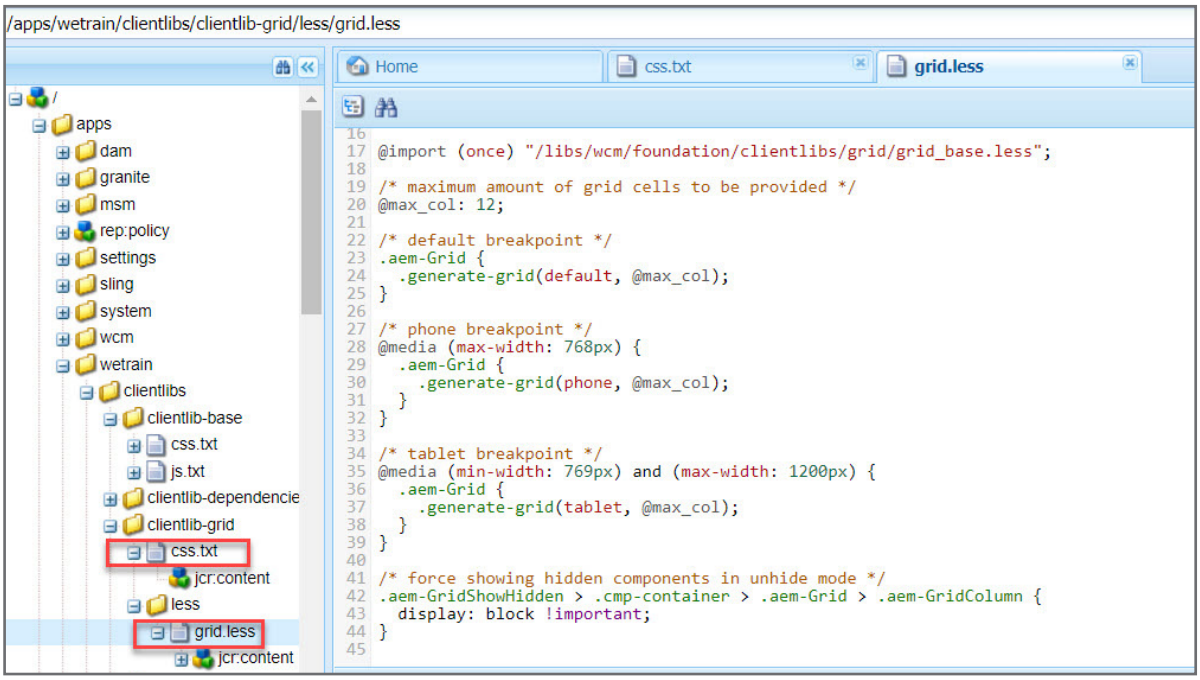
1. In your browser, open the AEM homepage and navigate to **Adobe Experience Manager** > **Tools** > **CRXDE Lite**. The **CRXDE Lite** page opens.
2. Navigate to the **/apps/wetrain/clientlibs/clientlib-base** folder and observe its properties: Looking at this client library, you can see these important nodes and properties:

   - Node type - cq:clientLibraryFolder
   - categories property – how the library is loaded in the web context.
   - allowProxy property – set to true due to common security settings in Production.
   - "manifest" child nodes of css.txt and/or js.txt.
   - Optional properties as needed – embed, dependencies, cssProcessor, etc…

If you open these libraries, you see that css.txt and js.txt in the clientlib-base are empty. clientlib-base is a 'hook' library. In this case, it has no actual Style nor JS code. It exists to load multiple other libraries on our website pages in a single call.
Because some of those other libraries attempt to load both CSS and JS, you need to have each manifest file type declared in clientlib-base.

These other libraries are loaded via the embed property -- in this case, several Core Component client libraries and the wetrain.grid library.

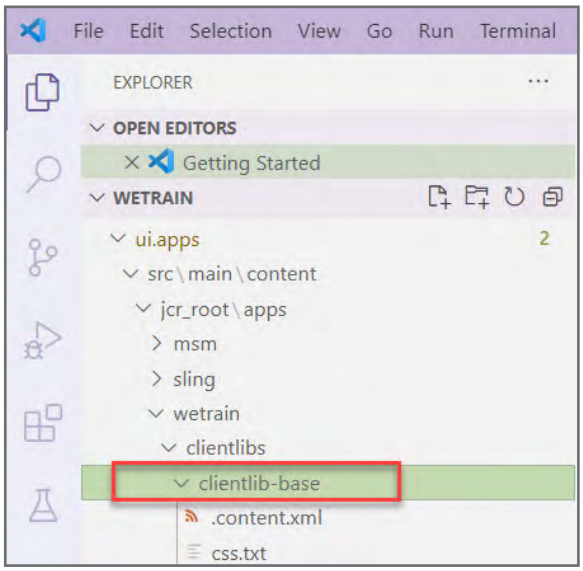
3. Navigate to **/apps/wetrain/clientlibs/clientlib-grid.** Expand **clientlib-grid** and open both the files **css.txt** and **grid.less** (double-click them) to view their contents, as shown:



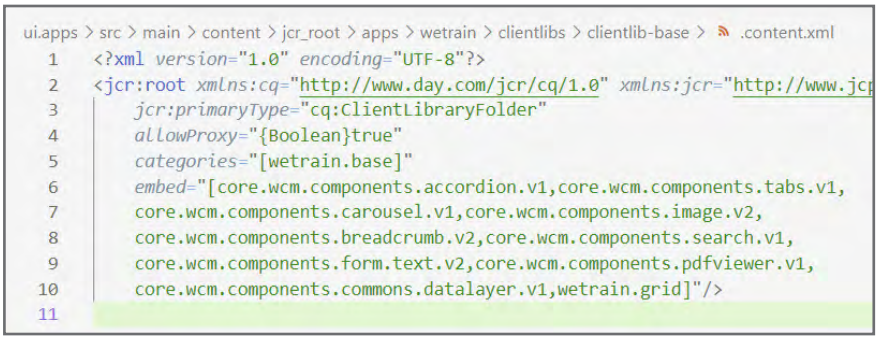
Note: css.txt loads grid.less, which is responsible for defining the device type breakpoints tracked by the Author environment's Layout mode.

To view the code the Archetype created in your project:

4. In your IDE, navigate to **ui.apps** > **src/main/content** > **jcr_root/apps** > **wetrain** > **clientlibs** > **clientlib-base**, as shown:



5. Verify the properties you viewed in **CRXDE Lite** exist in the .content.xml file:



You also see the .txt manifest files here.

Note: You will see similar folders under **clientlib-grid** which includes the actual grid.less file. Any needed changes can be made here and then deployed to the local quickstart. These libraries are built by the Archetype and packaged into the wetrain.ui.apps content package.

Note: When you build Style Code for your project in this way (files under ui.apps) you can use .css or .less files only – not .scss. The author service has a compiler for .less files, but not one for .scss.
To use .scss, we can leverage the ui.frontend Maven module to compile stylesheets outside of the author service and automatically add them to the wetrain.ui.apps content package.

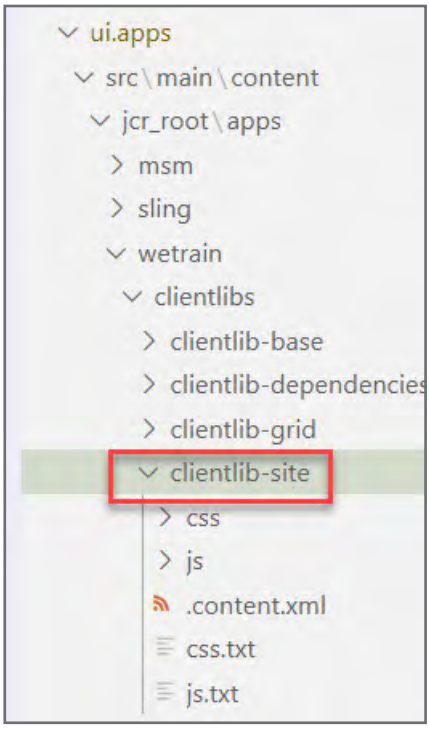
Exercise 2: Investigate the ui.frontend Maven Module

Front-end developers and designers typically use external tools to build and develop their design free of the hosted solution. The ui.frontend module is a module in which front-end developers can design their stylesheets and JavaScript externally, and also build and install into the author service with little knowledge of client libraries. In this exercise, you will investigate the ui.frontend module.

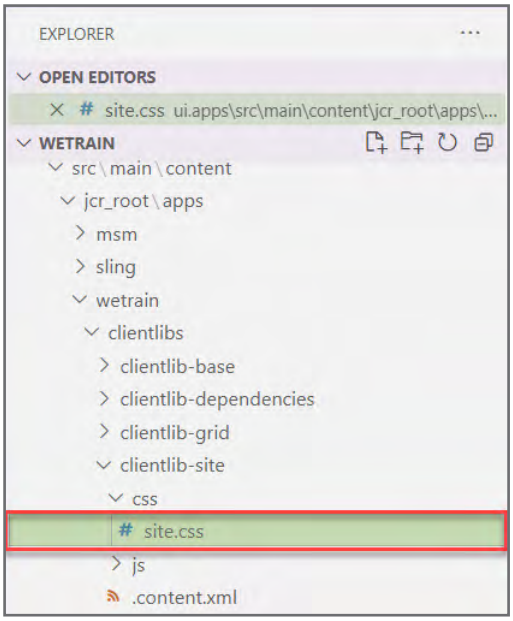
Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service

1. In your IDE, navigate to **ui.apps > src/main/content > jcr_root/apps > wetrain > clientlibs > clientlib-site.**
2. Investigate its structure and the contained files:



3. Double-click **site.css**, and examine the content of the file as shown:

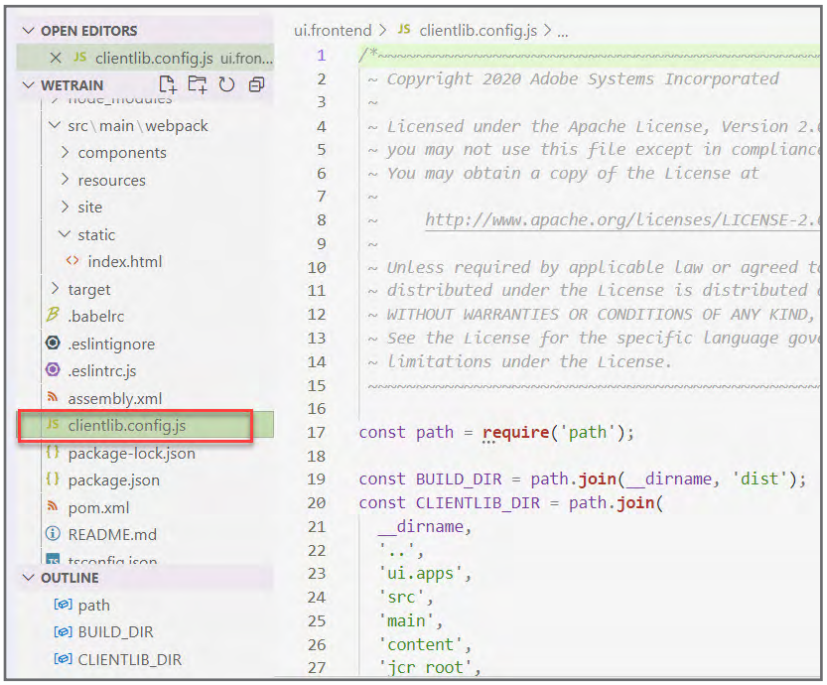


4. Similarly open **site.js** and examine the file.

Note: These are the two files produced by ui.frontend compilation.

5. Open **.content.xml.** Notice that the properties of this client library are also defined in ui.frontend in the clientlib.config.js file:
   a. categories, dependencies, allowProxy should all be familiar.
   b. cssProcessor and jsProcessor have to do with client library optimization. However, when we use ui.frontend Webpack optimizes our files, so the archetype "turns off" optimization with these settings of 'none'.

6. Navigate to the **ui.frontend** sub-module, expand it, and open **clientlib.config.js:**

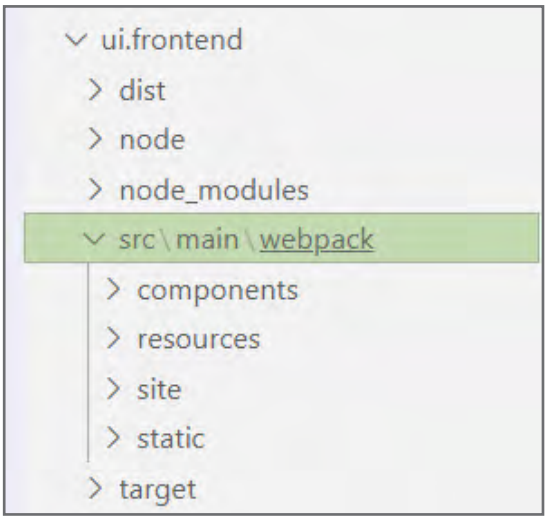


This file defines the node name and properties of the finished client library you just viewed in ui.apps. It is part of the npm plugin named aem-clientlib-generator.
At a high level, the files you code under **ui.frontend/src/main/webpack** and its subdirectories are compiled into ui.frontend/dist which are then built into the clientlib-site library in ui.apps.

7. View the files **webpack.dev.js** and **webpack.prod.js** further down the project.

   These two files allow us to manage development and production environment build settings separately.

8. Go to **ui.frontend** > **src/main/webpack** and view its folders:



- **components** - our Component code:
  - › Archetype starts out with an empty .scss file for each of our proxy components
  - › You will add more organization as you build out Component styles.
- **resources** - design related resources.
- **site**:
  a. For rules on what files webpack will load (main.ts & main.scss), notice that ALL .js and .scss files in the project will be loaded with the default settings - no need to granularly manage .scss partials with @import statements
  b. Reusable settings in _variables.scss.
  c. Any site wide CSS or JS code:
    i. Both _base.scss & _variables.scss.
    ii. stylesfolder:continer_main.scss,experiencefragment_footer.scss and experiencefragment_header.scss.
    iii. All contain example starter styles that can be applied via Policy settings.
- **static** folder is for a static index.html file. This enables development against a locally run webpack server to quickly view the .css or .js changes.

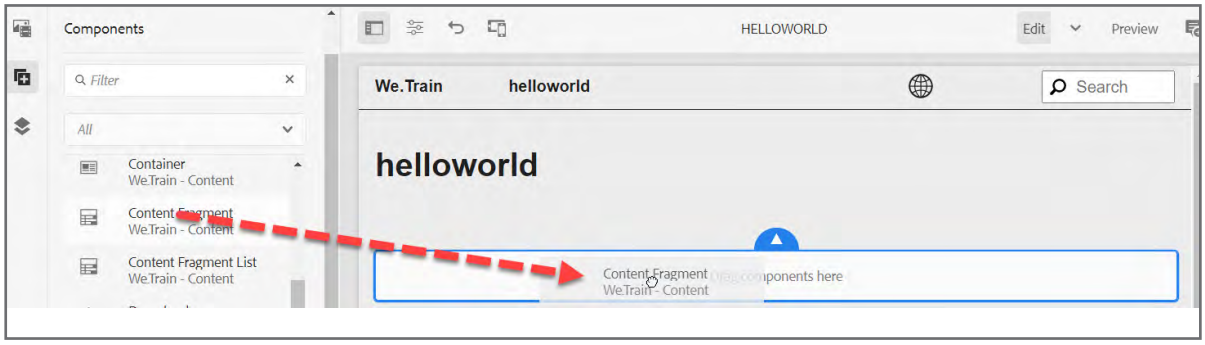
Exercise 3: Add Content to a Page

In this exercise, you will add some content to a page. After completing this exercise, you will see the content without a design. In later exercises, you will apply a design and you will see the content transform to the design standards of your project.

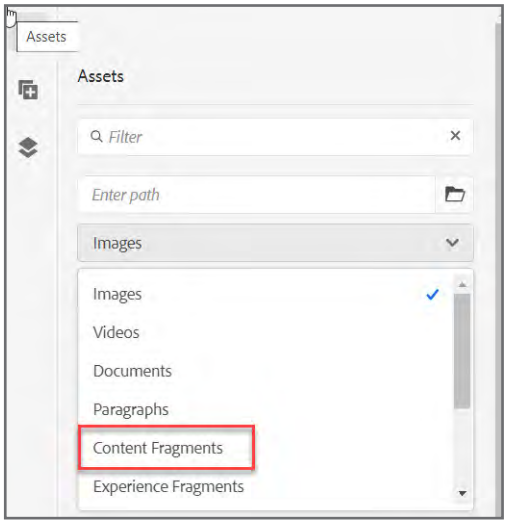
Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service

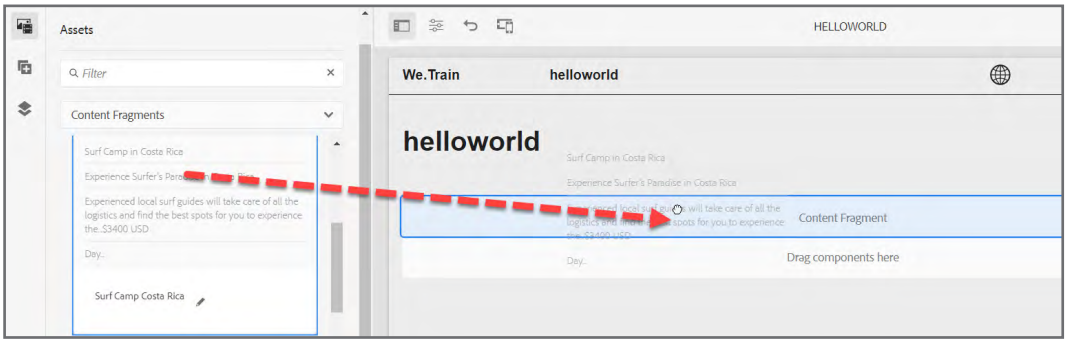
1. In your author service, navigate to **Sites** > **We.Train** > **Language Masters** > **en** > **helloworld** page.
2. Click the **helloworld** page checkbox and click **Edit (e)** on the header bar to open the **en** page.
3. On the left panel, click the **Components** icon. The available components are displayed.
4. Drag the **Content Fragment** component onto the layout container, as shown.



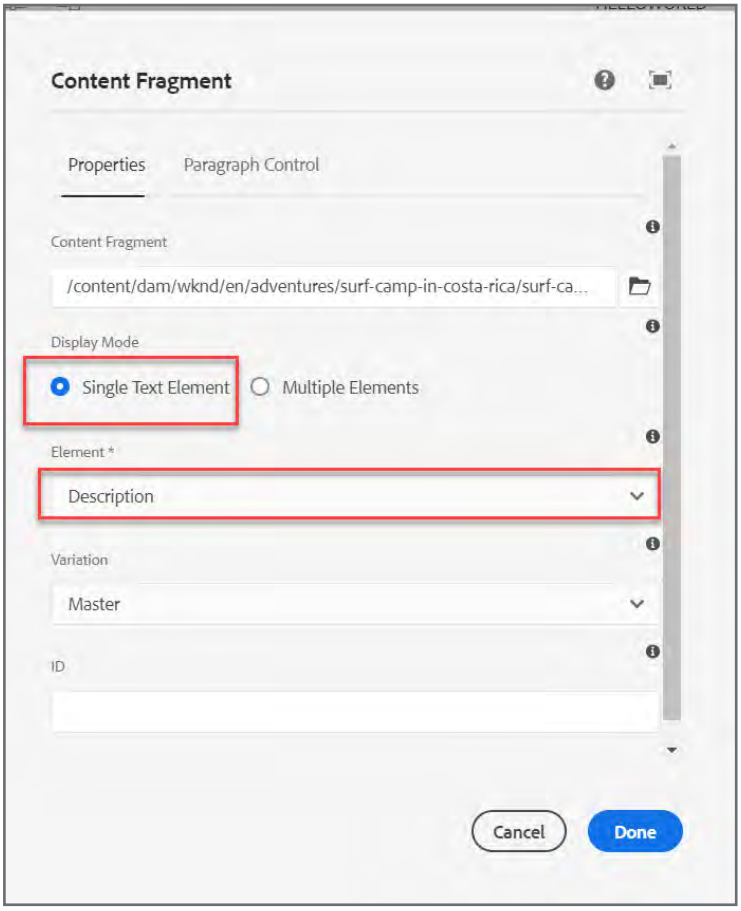
5. On the left panel, click the **Assets** icon. The available assets are displayed.
6. Filter the assets based on Content Fragments by clicking the **Images** dropdown menu and selecting **Content Fragments**, as shown:



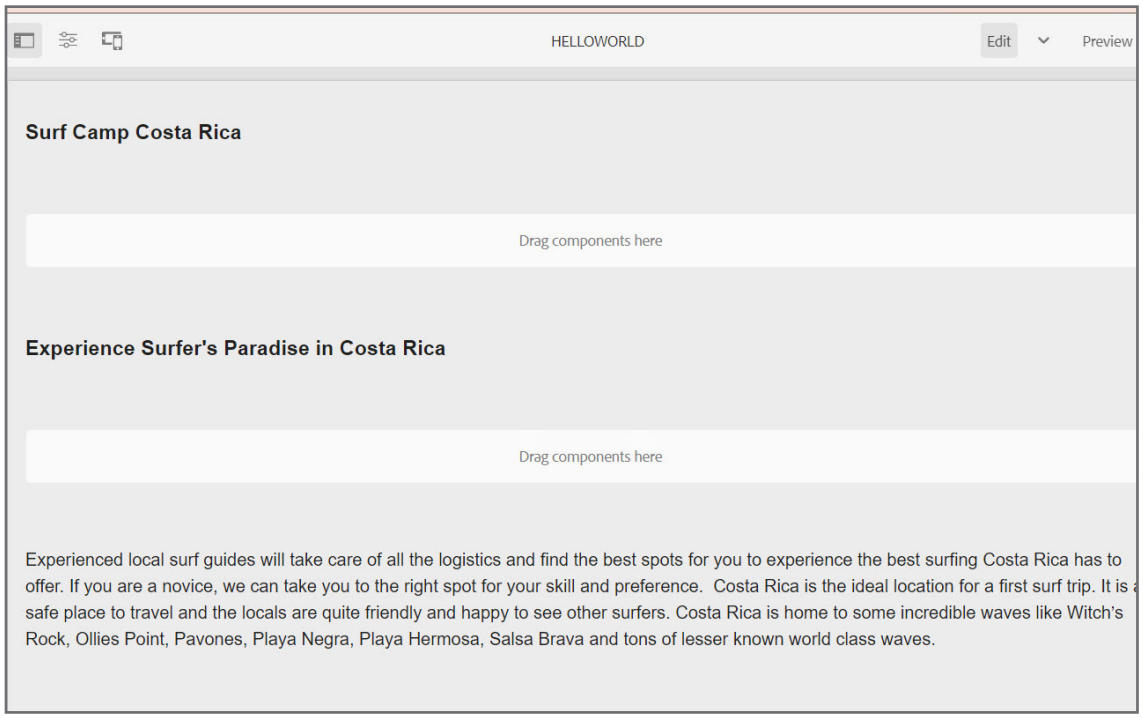
7. Drag the **Surf Camp Costa Risurfca** fragment onto the Content Fragment component, as shown.



8. Notice how the content has no design currently.
9. Click the **Content Fragment** and select the Configure icon (wrench icon). The **Content Fragment** dialog box opens.
   - Select **Single Text Element** for the **Display Mode**.
   - From the Element dropdown menu, select **Description**.



10. Click **Done**. Notice that you can now view the description in the Content Fragment.



Exercise 4: Set up Webpack for Development

In this exercise, you will start a Webpack server and copy the HTML of an AEM page to use with Webpack.

Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service

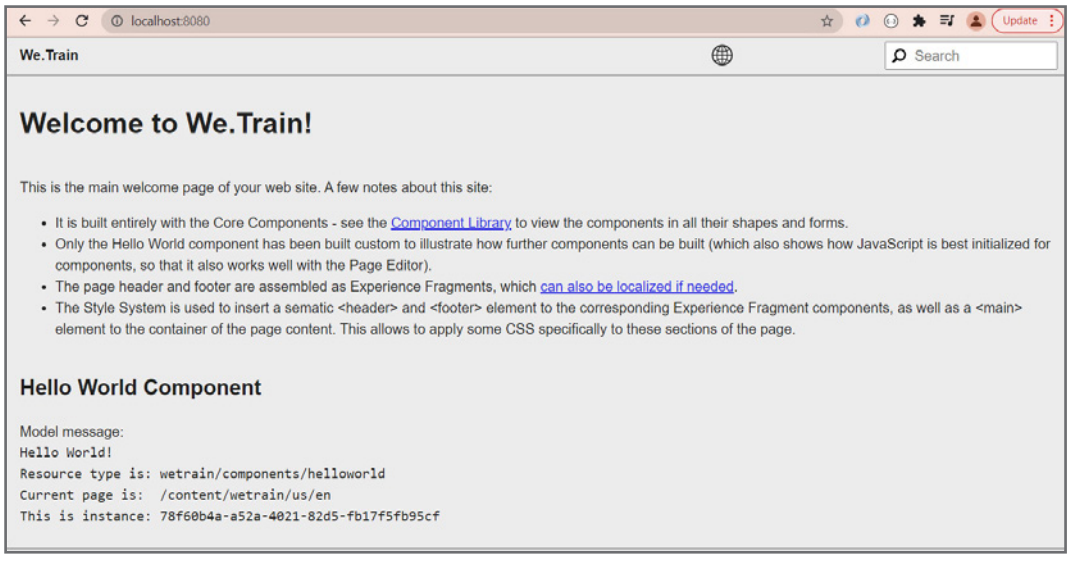
1. In your IDE, open a terminal window.

Note: As an alternative to the IDE terminal window you could open an OS terminal window in your <AEM Project> folder.

2. In the terminal window, cd to **ui.frontend** directory.

3. Type in the following command to start the Webpack server. The Webpack server is started, as shown:

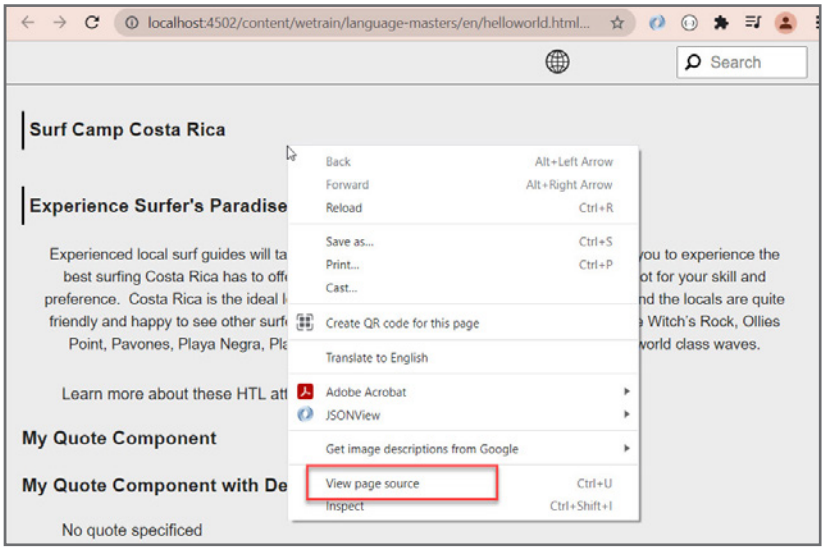
```
npm run start
```



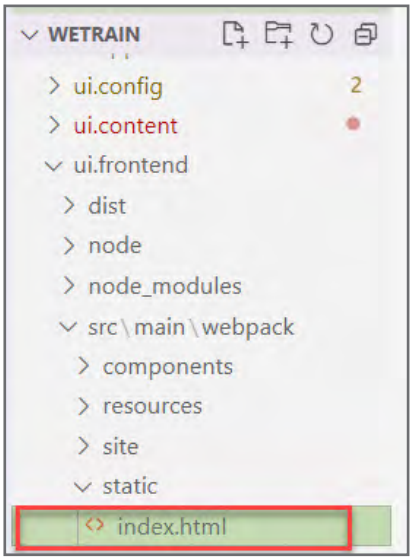
Note: Webpack server is started and running on localhost:8080. It is best to view localhost:8080 on the same browser where you are logged into AEM. The web content you see at **8080** is the default **index.html** file provided by the Archetype.

To replace the default web content:

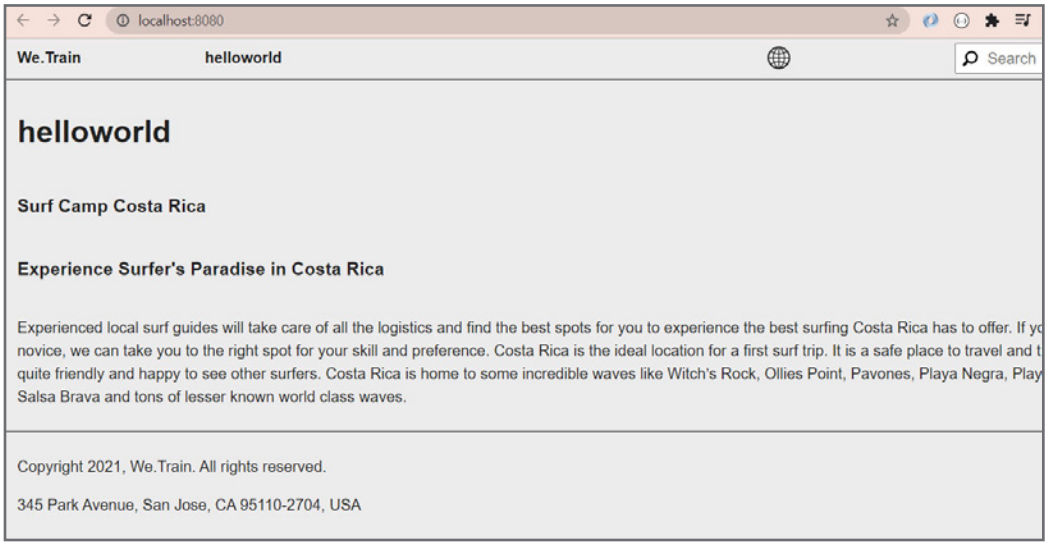
4. In a browser, open the Sites console and navigate to the **We.Train** > **Language Masters** > **en** > **helloworld** and open it in the authoring UI.
5. Click **Page Information** > **View as Published**. The page opens on a separate tab of the browser.
6. Right-click on the page and click **View page Source**, as shown:



7. Copy all the source html.
8. In your IDE, paste the copied source html into the **wetrain/ui.frontend/src/main/webpack/static/index.html** file, replacing all initial content in **index.html**.



9. Save the changes (**File** > **Save**).
10. In the Browser tab showing 8080, you should now see our Hello World page with Content Fragment and Quote components, as shown:



Note: Now that we have set up the Webpack server, the changes in the following exercise will automatically be visible on 8080 as opposed to having to run a build and deploy to the local quickstart.

Note: If you are receiving any errors while trying to start the Webpack server, the npm version used could be old for your ui.frontend npm project. Archetype 39 should use a minimum of npm v18 to run successfully. In Readytech you can check/change the version by following these steps:

1. Open a command prompt as an Administrator.
2. View installed npm versions: **nvm ls**
3. Change npm version: **nvm use 18.11.00**

Exercise 5: Create a Design for a Component

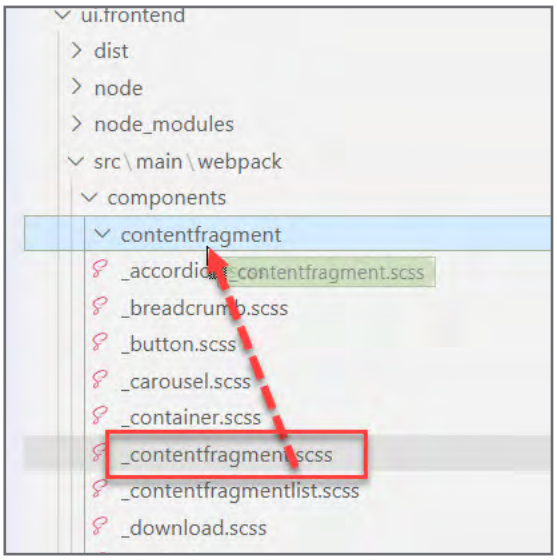
In this exercise, you will create a design for a component. To follow best BEM practices, we will start style development by creating better organization in our ui.frontend/src/main/webpack/components folder.
In this exercise you will create a design for a component, then build and deploy to the local quickstart.

Prerequisites:

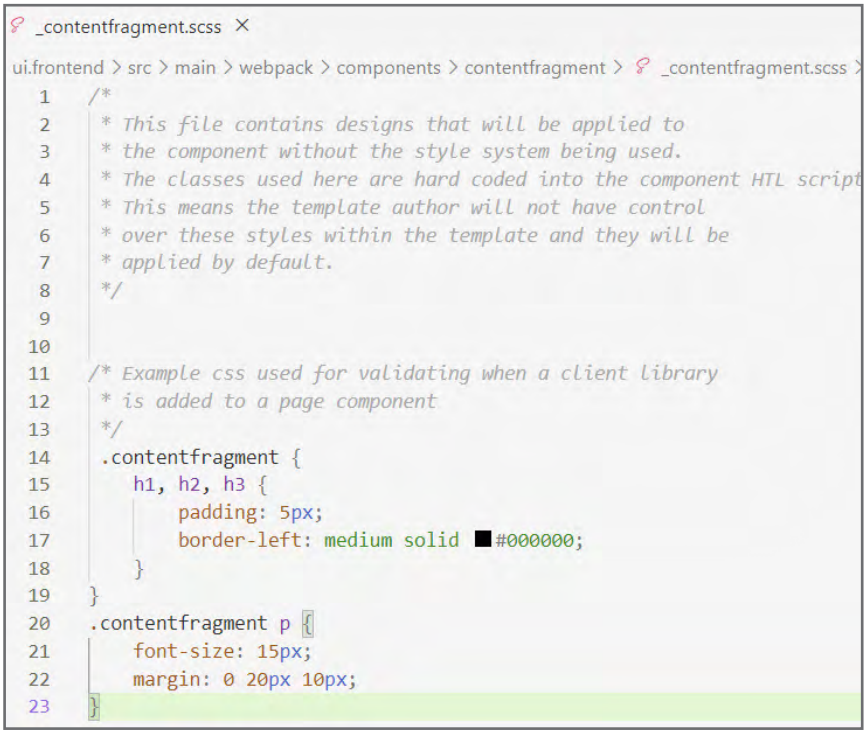
- Local author service running
- A Maven project imported into your IDE and installed on the author service

Task 1: Create a Design for a Component

1. Using your IDE, navigate to the **ui.frontend/src/main/webpack/components** folder.
2. Right-click the folder **components** and choose **New Folder.**
3. Name the folder **contentfragment.**
4. Drag and drop the existing **_contentfragment.scss** file into this new **contentfragment** folder, as shown:



5. In your **Exercise_Files-DWC** folder, navigate to **ui.frontend\src\main\webpack\components\contentfragment** and copy the contents of the **_contentfragment.scss** file.
6. In your IDE, replace all current content in the **ui.frontend\src\main\webpack\components\contentfragment_contentfragment.scss** file with the copied content.

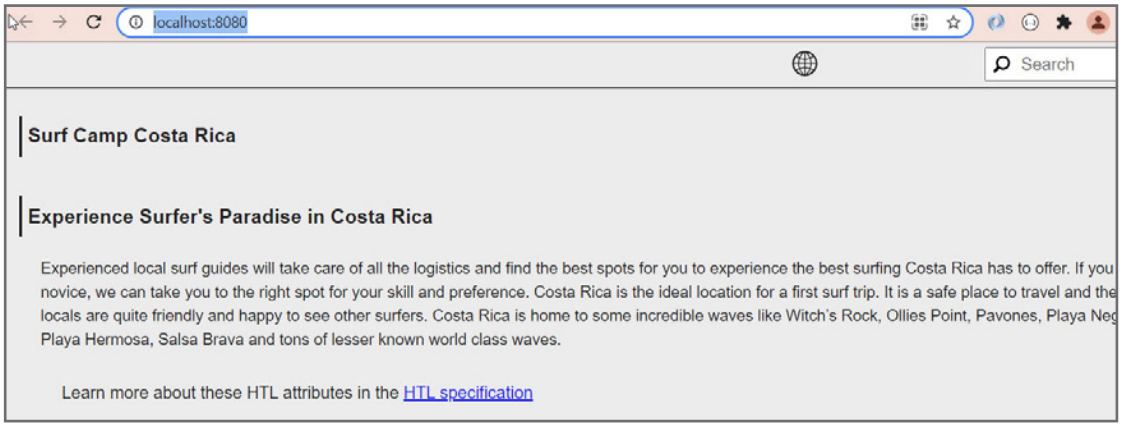


7. Save the changes.

Note: In this approach , we are taking a view that the default style for each of our components will be built in a .scss file that shares the component's name. Additional styling options (which we will look at later) will be built in a file named for the given styling option (and applied via the Style System) .
This is NOT the only valid approach to code organization nor is it required for the project to work correctly. For example - you might have your default style rules in a file called 'default.scss' and then build variant styling into other files.

---

8. If you are working with the Webpack development server, you now see these styles applied on the localhost:8080 view, as shown:



Note: If you do not see your changes, your browser may be caching and you will need to use a hard refresh. In Chrome, go to Preview Mode and right-click the page, inspect and then right-click hard refresh. Empty cache from reload icon to the left of the address bar.

9. Stop the Webpack server by pressing <Ctrl+C> in the Terminal window.

You may skip the next task if you are working with the Webpack development server.

Task 2: Build and Deploy to the Local Quickstart

1. Navigate to a terminal window with your <AEM Project> in it.

Note: As an alternative to the IDE terminal window you could open an OS terminal window in your <AEM Project> folder.

2. In the terminal window, cd to **ui.frontend** directory.

3. Type in the following command to compile the ui.frontend sub-module, as shown:

```
npm run dev
```

4. Navigate to **ui.apps** and view the **clientlib-site** library. Open th**e site.css** file and see that the contentfragment rules have been added (starts around line 43).
   Notice that they are not deployed to the local quickstart yet.

5. In the VS Code Terminal, type: **cd ../ui.apps** then press <Enter> which takes us back to the **wetrain/ui.apps** folder level.

6. Run the command:

```
mvn clean install -PautoInstallPackage
```

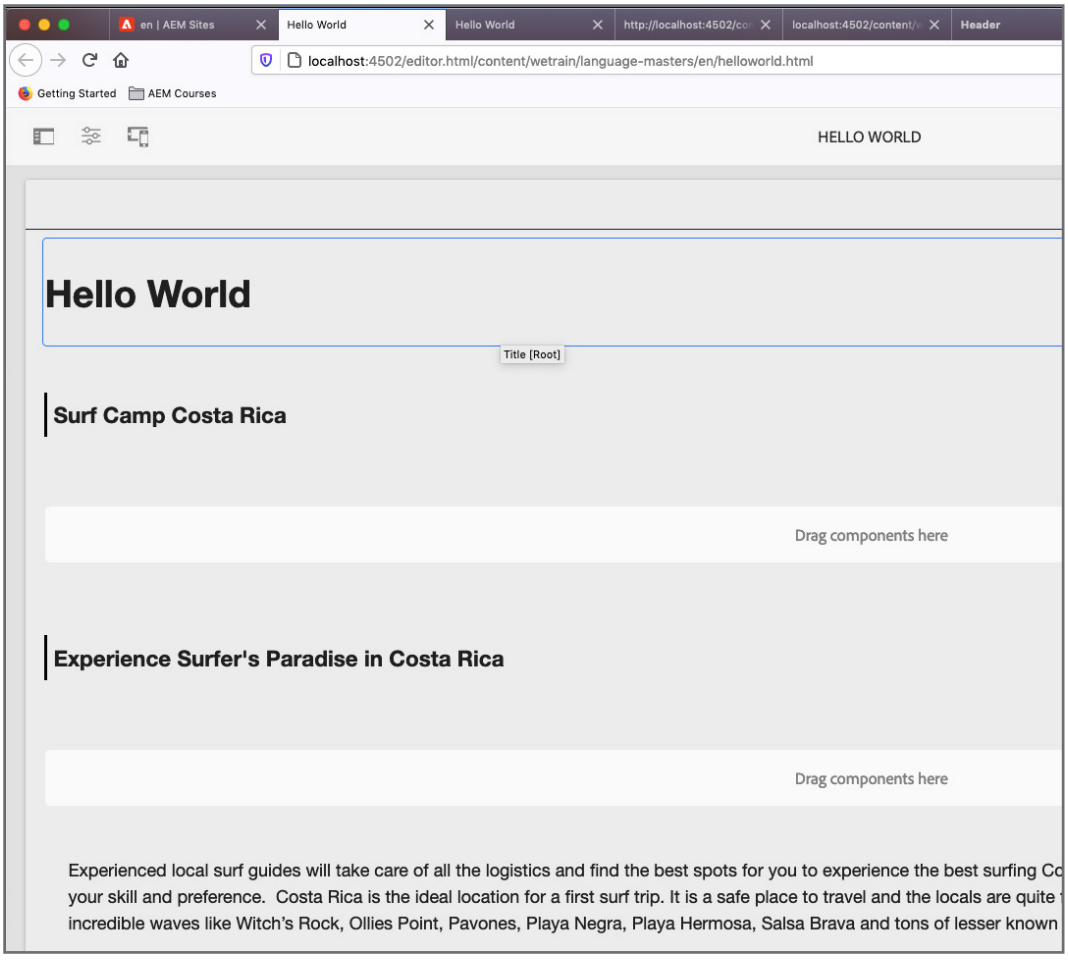
This builds only the ui.apps submodule and deploys only the ui.apps content package to the local quickstart.

Note: Notice it builds much faster than a build of the entire project, which is why we show this two-step approach:

1. Build ui.frontend
2. Build ui.apps

---

7. In a browser, go back to the tab with Helloworld page and refresh or reopen it in the Sites console: **We.Train** > **Language Masters** > **en** > **helloworld**. You should see your style changes applied, as shown:



Exercise 6: Add a Page Style Using the Style System

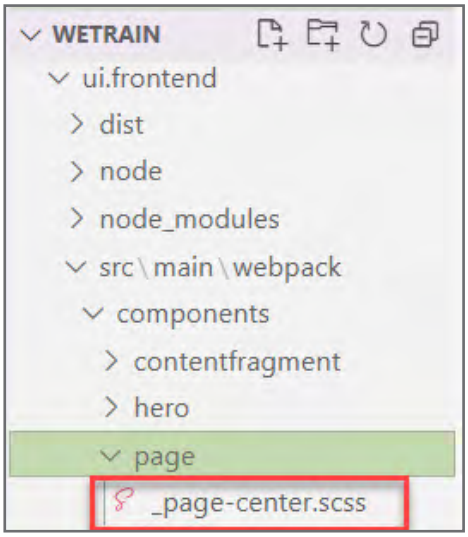
At times, an author needs to incorporate an optional design in a page. You can provide this flexibility to an author with the Style System in content policies. In this exercise, you will create a style and apply it at the page level.

Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service

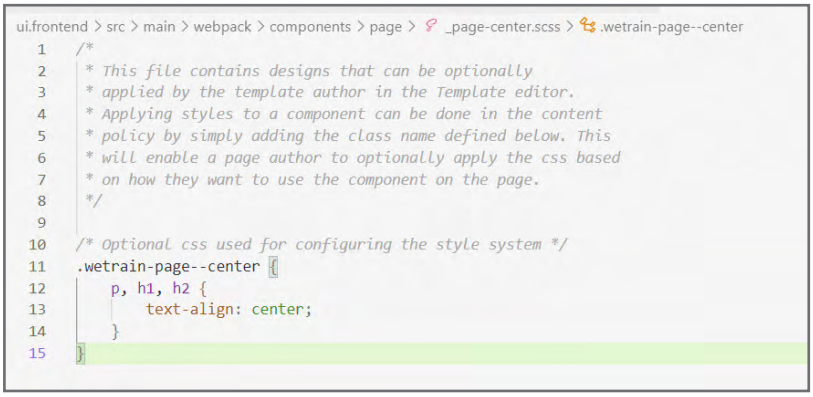
Task 1: Create a Style

1. Using your IDE, navigate to the **ui.frontend/src/main/webpack/components** folder.
2. Right-click the folder **components** and choose **New Folder.**
3. Name the folder **page.**
4. Right-click the folder **page** and choose **New File.** Name the file **_page-center.scss, a**s shown:



5. In your **Exercise_Files-DWC** folder, navigate to **ui.frontend\src\main\webpack\components\page** and copy the contents of the **_page-center.scss** file.

6. In your IDE, paste the copied content in **ui.frontend\src\main\webpack\components\page_page-center.scsss** file with the copied content.



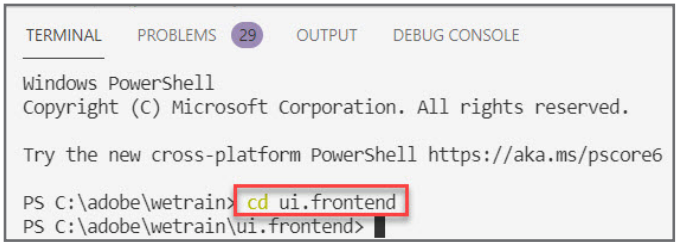
7. Save the changes.

Task 2: Deploy the Project

1. Using your IDE, open a terminal window.

Note: As an alternative to the IDE terminal window you could open an OS terminal window in your <AEM Project> folder.

2. In the terminal window, cd to the **ui.frontend** directory.



3. Run the following command to compile the ui.frontend sub-module, as shown:

```
npm run dev
```

4. Once the build finishes, navigate to the **ui.apps** module:

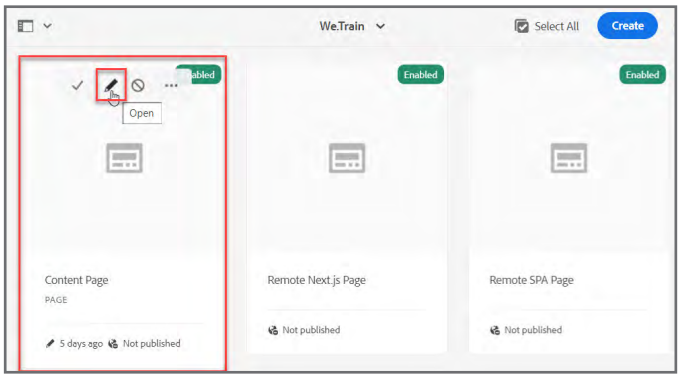
```
cd ../ui.apps
```

5. Run the following command and verify it ran successfully:

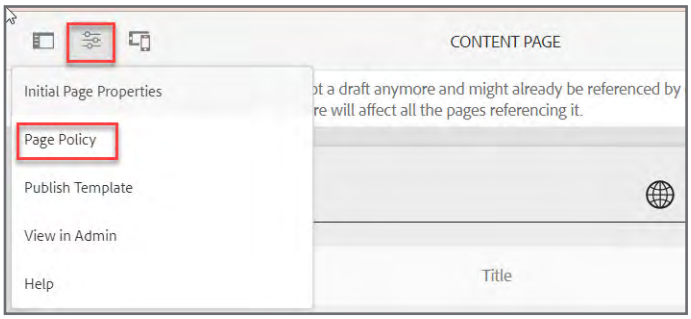
```
mvn clean install -PautoInstallPackage
```

Task 3: Apply the Style at the Page Level

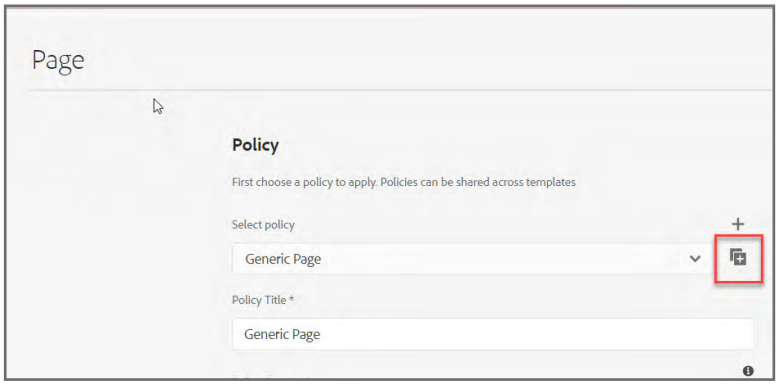
1. In a browser, open AEM home page and navigate to **Adobe Experience Manager** > **Tools** > **Templates** > **We.Train.**
2. Click the **We.Train** folder to open it.
3. Hover the cursor over the **Content Page** template and click the Open icon (pencil icon). The Template editor opens.



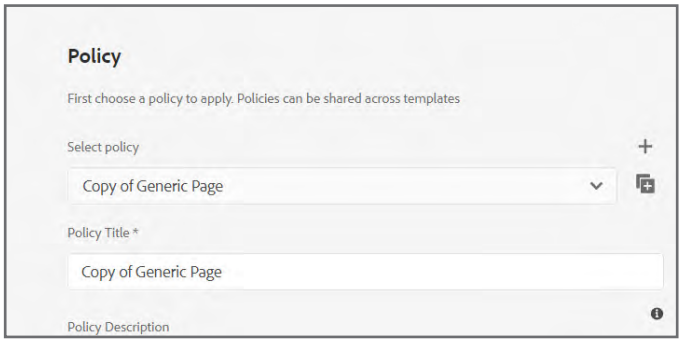
4. Click **Page Information** > **Page Policy**, as shown. The Page wizard opens.



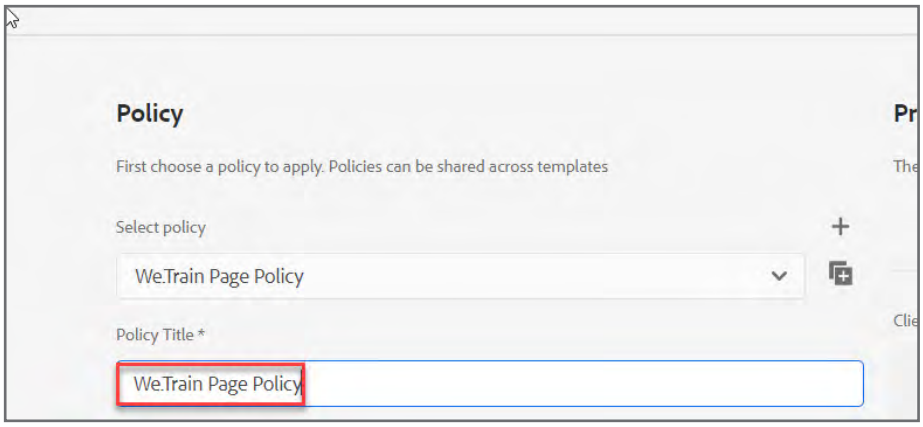
5. Copy the **Generic Page** policy, as shown:



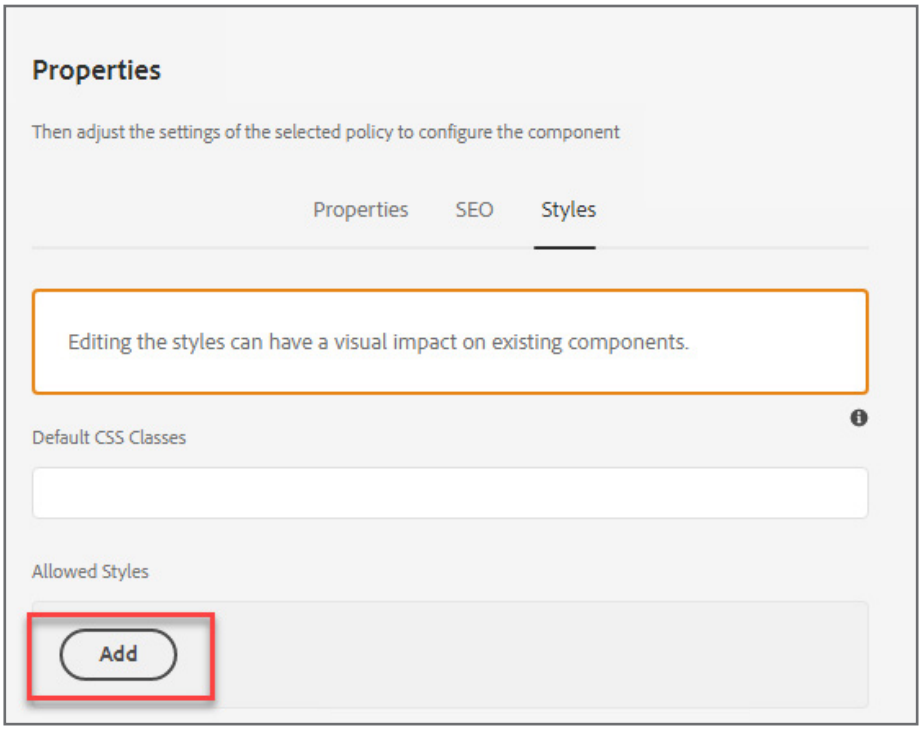
A copy of Generic Page is created.



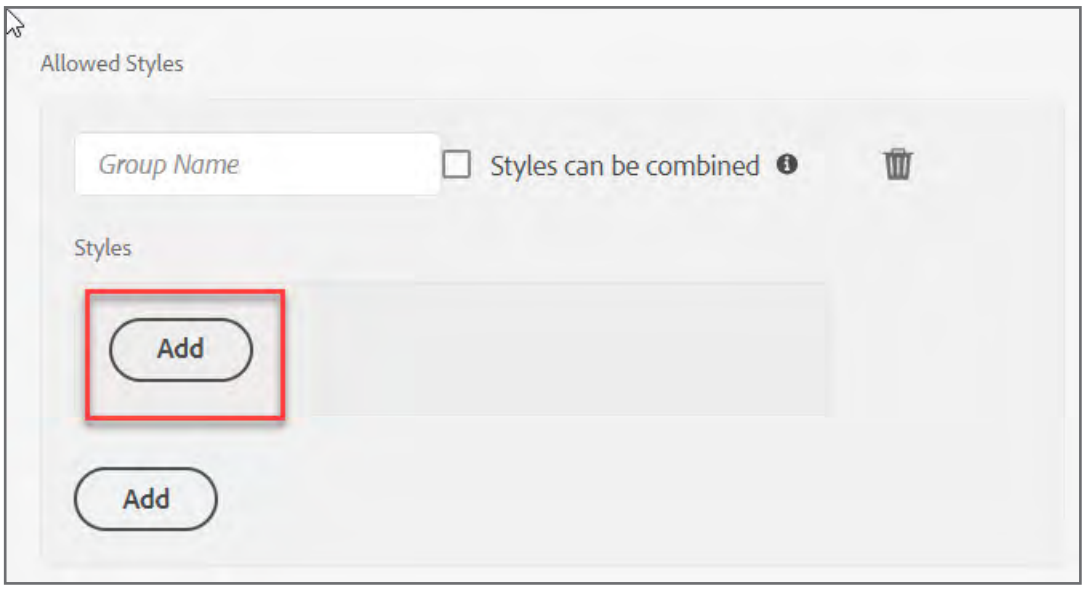
6. Rename the copied policy **We.Train Page Policy**, as shown:



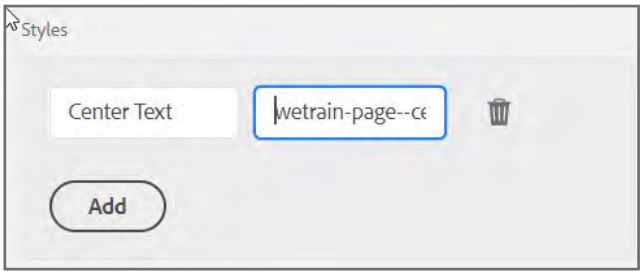
7. In the **Properties** section, on the **Styles** tab, click **Add** under **Allowed Styles,** as shown. The Styles section appears.



8. Click **Add** under **Styles**, as shown. The **Style Name** and **CSS Classes** boxes appear.

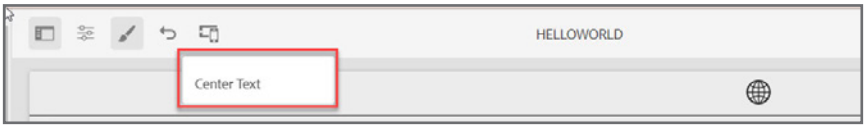


9. In the **Style Name** box, type **Center Text.**
10. In the **CSS Classes** box, enter **wetrain-page--center**

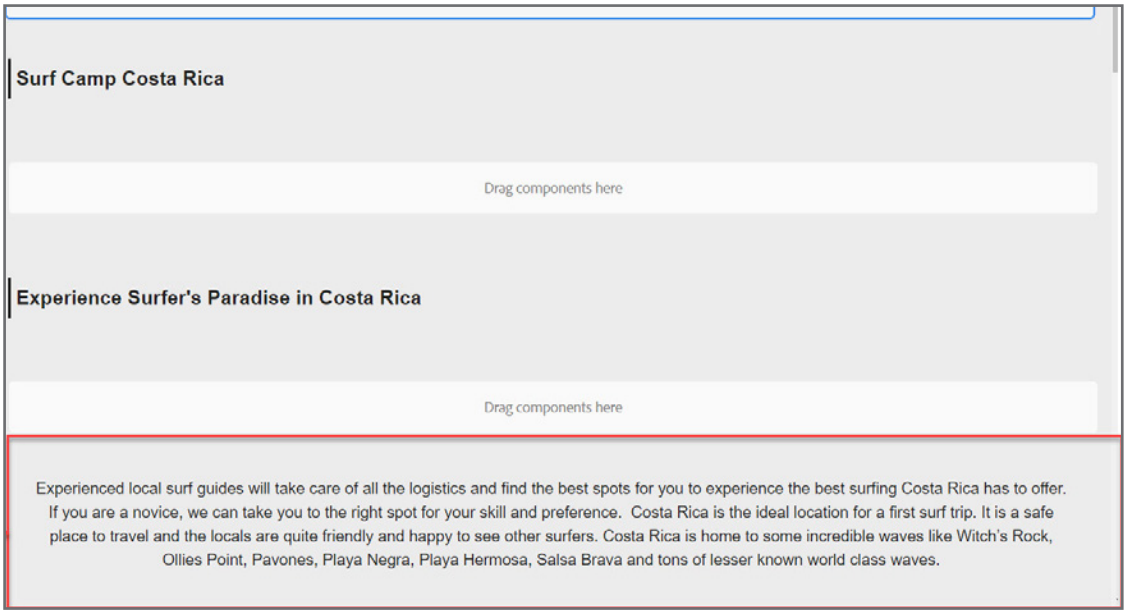


11. Click **Done.**
12. Use a browser to navigate to **Sites** > **We.Train** > **Language Masters** > **en** > **helloworld**.
13. Open the **helloworld** page in the page editor.
14. Notice there is a Styles icon (paintbrush icon) on the toolbar, as shown:

15. Ensure the Content Fragment is selected, click the Styles icon and select Center Text, as shown:



16. Notice that the description in the Content Fragment is center aligned now.



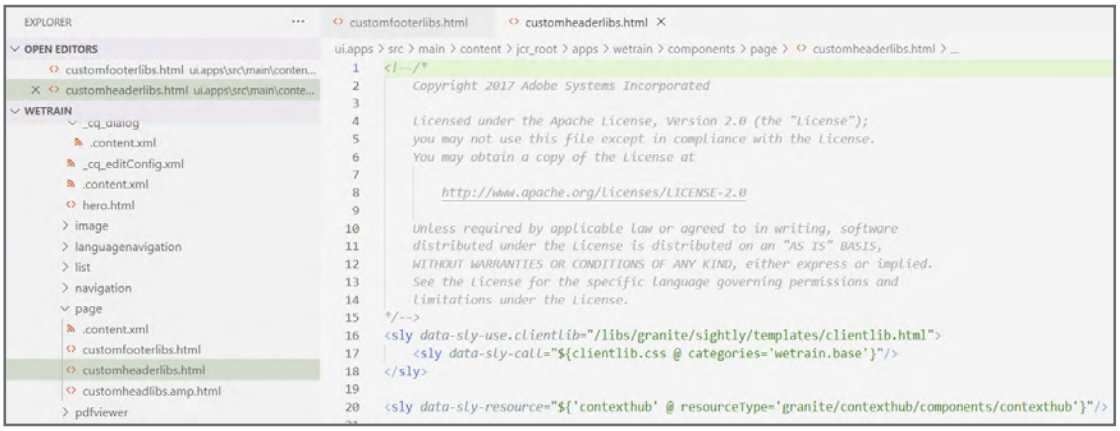
Exercise 7: Investigate the Page Component Inclusions

As a developer, you may want to load client libraries in a way that cannot be edited/altered by the Authoring UI. In these cases you will want to include via HTL. The Archetype does this for us with the typical libraries needed to start a project.
In this exercise you will create a style and apply at the page level, then deploy and verify.
Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service

Task 1: Create a Style and Apply at the Page Level

1. In your IDE, navigate to the **ui.apps** > **src/main/content** > **jcr_root/apps** > **wetrain/components/page** folder.
2. Select **customheaderlibs.html** under the **page** folder to open in the IDE editor:



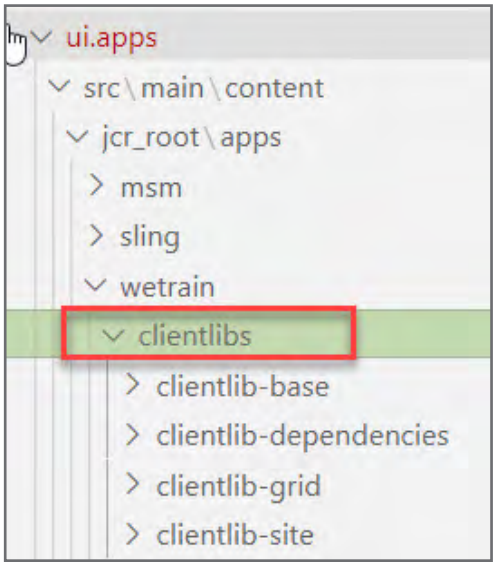
Verify the inclusion of client libraries from your wetrain.base library around line numbers 16-18. Notice the CSS loaded in customheaderlibs.html.

3. Open **customfooterlibs.html** files in the IDE editor and verify the inclusion of code from your wetrain.base library around line numbers 16-18.

```
16    <sly data-sly-use.clientlib="/libs/granite/sightly/templates/clientlib.html">
17        <sly data-sly-call="${clientlib.js @ categories='wetrain.base'}"/>
18    </sly>
```

Notice the JavaScript loaded in **customfooterlibs.html**.

4. Scroll up to the j**cr_root/apps/wetrain/clientlibs** directory, as shown:



5. Expand **clientlib-base** and open the **.content.xml** file.

```
ui.apps > src > main > content > jcr_root > apps > wetrain > clientlibs > clientlib-base > .content.xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <jcr:root xmlns:cq="http://www.day.com/jcr/cq/1.0" xmlns:jcr="http://www.jcp.org/jcr/1.0"
3      jcr:primaryType="cq:ClientLibraryFolder"
4      allowProxy="{Boolean}true"
5      categories="[wetrain.base]"
6      embed="[core.wcm.components.accordion.v1,core.wcm.components.tabs.v1,core.wcm.components.carousel.v1,core.wcm.components.
7
```

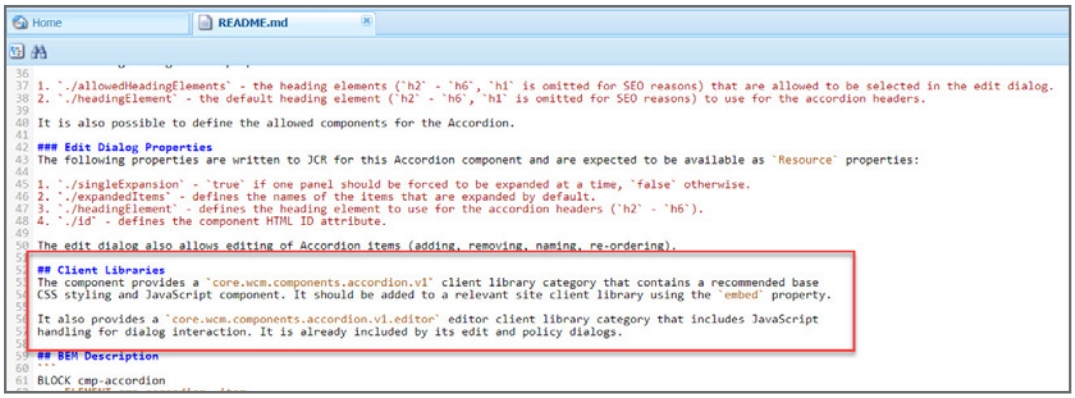
This is the library that the header and footer libs are loading along with all the embedded libraries defined in this file.

6. Notice all the embedded Core Component libraries:

Note: When you proxy a Core Component through the sling:resourceSuperType property, this does not load the client libraries. If you need them loaded, you have to bring them in through the clientlibs convention you see here.

7. In **CRXDE Lite**, open the accordion Core Component
   [/libs | /apps]/core/wcm/components/accordion/v1/accordion

8. Open the **README.md** file and find the section ## Client Libraries section.



Note: When a Core Component has clientlib code that we need to include in our project, it is specified here. Archetype starts us off with the ones you need today. You can add any future additions to our project code if needed.

Here is a view on how the browser loads this library's css (js is similar, down near the end of a given webpage):

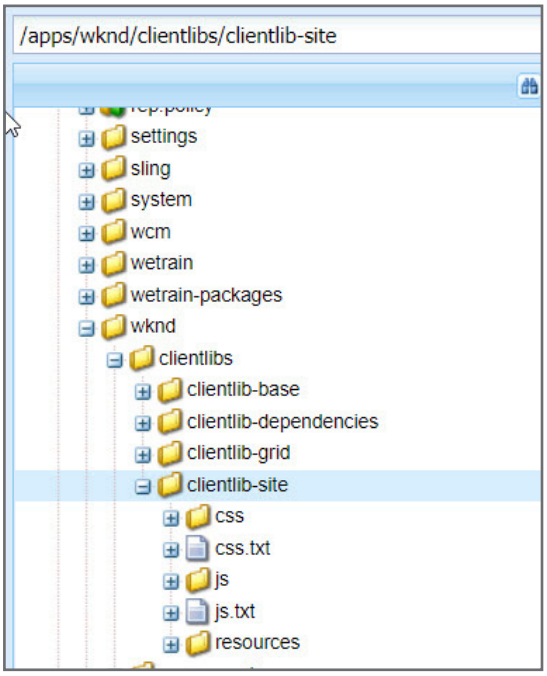
The /etc.clientlibs/.. portion is added due to our allowProxy property being set to true, which accommodates common production settings that restrict access to /apps.

Exercise 8: Include Client Libraries via Template Policies

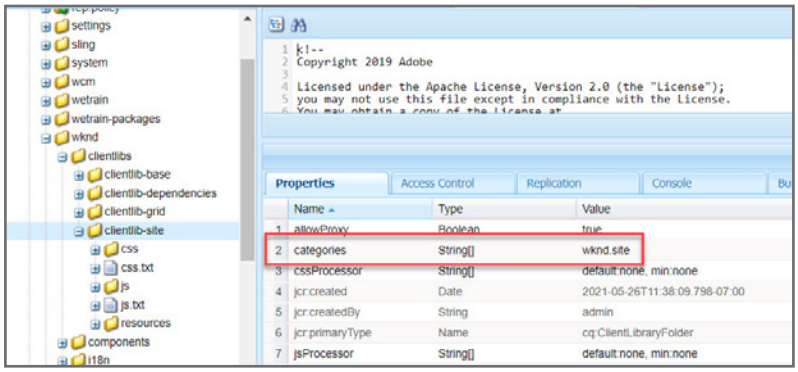
In certain cases, you might want to switch the 'theme' of a website without needing to run through a full code deployment, or you might want to use another project's client library that already exists on the author service. The Policy of an Editable Template allows a way to achieve either. We will look in full detail at Editable Templates and Policies in a later module.
In this exercise, we will locate the provided location for client library inclusion, and we will bring in the bulk of styles for We.Train from the existing WKND client libraries.
Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service

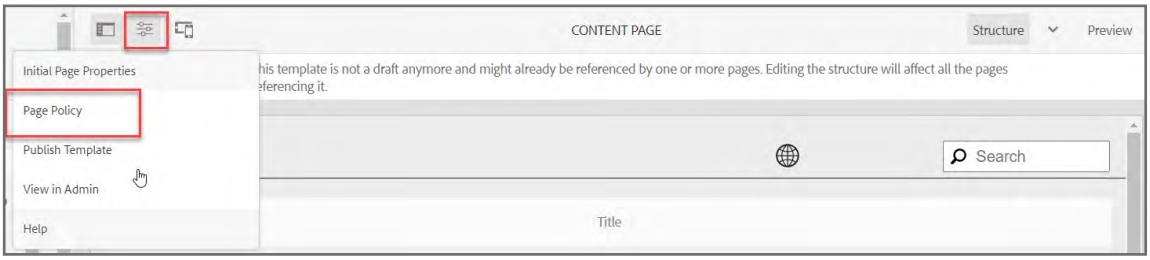
1. In **CRXDE Lite** and navigate to **/apps/wknd/clientlibs/clientlib-site**.
   This is the compiled client library with WKND style code, which you will add to We.Train:



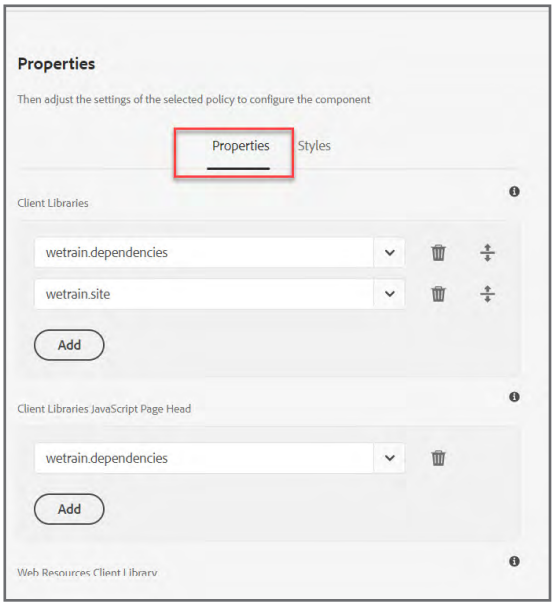
2. Notice the categories property value **wknd.site.** You will include the library in your template using the category name **wknd.site**.



3. In your browser, open AEM home page and navigate to go to **Tools** > **General** > **Templates** > **We.Train.**
4. Hover over the **Content Page** template and select Edit (pencil icon).
5. On the resulting screen, select Page Info icon then **Page Policy**, as shown:



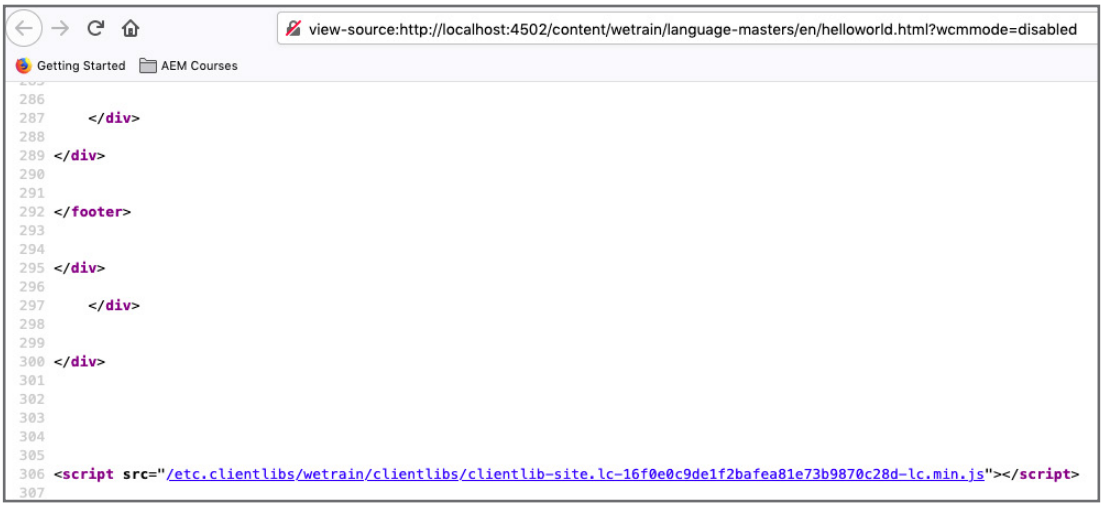
6. Notice the **Properties** section on the right of the screen. This is how the style code we built earlier is being loaded on the page. The Archetype establishes these settings from the start.



In the first section, we name client library(s) to load. (This setting passes down to all Pages built from this Template.) We are not using any dependencies in our project currently, but if we were, they would be loaded.

AEM automatically sets any CSS in the named libraries in this section to load early in page rendering, in the <head>:

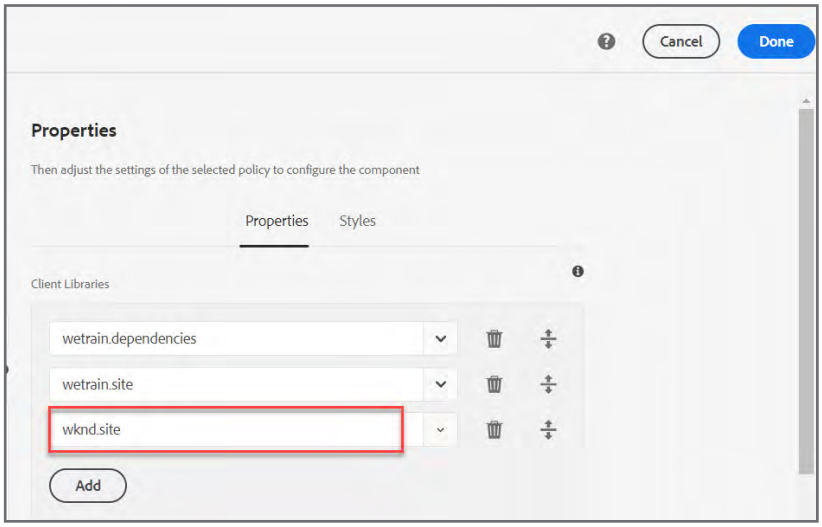
Any JS in this section loads near end of page rendering, after the <footer>:



For JavaScript, you specifically need to load in <head>. The next section of this Properties panel, labeled **Client Libraries JavaScript Page Head,** allows you to do so:

Notice via the 'information' icon that if this library also has CSS you want to load. You need this library to be named in the top level "Client Libraries" section, as shown below:

7. To view the results of our next steps, make sure you have a browser tab open to our helloworld page in AEM. Do not navigate away from this screen of the Page Policy.
8. Back in the browser tab open to the Page Policy, in the top **Client Libraries** section of Properties, select **Add** and add **wknd.site** library, as shown:



9. Click **Done** in the top right corner to save the change.
10. Go to the browser tab of the helloworld page and notice the visual appearance of the content.
11. Refresh the helloworld page and you will notice some changes to our content due to the WKND style rules applied.

References

- More details on JS & CSS optimization options here:
  https://experienceleague.adobe.com/docs/experience-manager-core-components/using/developing/archetype/uifrontend.html?lang=en#output
- AEM Project Archetype Front-End Build
  https://experienceleague.adobe.com/docs/experience-manager-core-components/using/developing/archetype/uifrontend.html?lang=en#possible-workflows

Extend Core Components

Introduction

Adobe Experience Manager (AEM) components are used to hold, format, and render the content on your webpages. Depending on the component you want to implement in your project, you can extend or customize an existing component, rather than defining and developing the entire structure from the beginning.

By extending the Core components you can create the desired custom functionality. The Core components implement several patterns that allow easy customization, from simple styling to advanced functionality reuse.

Objectives

After completing this module, you will be able to:

- Extend a core component
- Customize the component dialog
- Configure drag-and-drop capabilities using cq:EditConfig
- Add a default design to a proxy component
- Create a Sling Model for headless output

Core vs. Extend vs. Custom

New components should extend Core components if at all possible. To ensure understanding of the Core components and how they can jump start your project, reference the Component Library in the design phase. The Core Components have been developed using Adobe best practices, are flexible and powerful, and can save a lot of work.

For more on developing core components, use the link in the **References** section.

Only add truly custom components when there is a real business need that cannot be reasonably achieved through the extension of a Core component. To know more on **Paths to Success with the Core Components** use the link in the **References** section.

Core Component Extension Patterns

Dialog Boxes

You may want to customize the configuration options available in a core component dialogs. Both edit and design dialogs can be customized. Each dialog has a consistent node structure. It is recommended that this structure is replicated in the child component that inherits from a Core component. Replicating the dialog node structure allows the Sling Resource Merger to merge the child dialog structure with the inherited core Component dialog structure. In addition, Hide Conditions can be used to hide, replace or reorder sections of the inherited dialog.

Customizing the Markup

Sometimes adding advanced styling or capabilities, like rendering additional field values, requires a different component script. This can easily be done by copying the HTL file(s) that needs to be modified from the Core Component into the proxy component.

Customizing the Logic of a Component

The Business logic for the core components is implemented in Sling Models. This logic can be extended by adding a new Sling Model, for the proxy component, that extends the inherited Sling Model.

Styling the Components

Core components follow a standardized naming convention to allow for easy styling. Each Core component is wrapped in a element with the "cmp" and "cmp-<component-name>" classes. To make sure a CSS rule only affects the proxy component class all rules should be namespaces. For example:

```
.cmp-hero   .hero {}
.cmp-hero   .hero-title {}
.cmp-hero   a {}
```

Each of the Core components leverage the Style System feature that allows template authors to define additional CSS class names that can be applied to the component by page authors.

cq:editConfig

The **cq:editConfig** node defines the edit properties of the component and enables the component to appear in the Components browser.

Note: If the component has a dialog, it will automatically appear in the Components browser, even if the cq:editConfig node does not exist.

The edit behavior of a component is configured by adding a **cq:editConfig** node of type **cq:EditConfig** below the component node (of type **cq:Component**) and by adding specific properties and child nodes. The following properties and child nodes are available:

- cq:editConfig node properties:
  - **cq:actions** defines the actions that can be performed on the component.
  - **cq:layout** defines how the component is edited in the classic UI.
  - **cq:emptyTextb** defines text that is displayed when no visual content is present.
  - **cq:inherit** defines whether missing values are inherited from the component that this component inherits from.
- cq:editConfig child nodes:
  - **cq:dropTargets** defines a list of drop targets that can accept a drop from the Asset browser.
  - **cq:actionConfigs** defines a list of new actions that are appended to the cq:actions list.
  - **cq:formParameters** defines additional parameters that are added to the dialog form.
  - **cq:inplaceEditing** defines an inplace editing configuration for the component.
  - **cq:listeners** defines what happens before or after an action occurs on the component.

cq:editConfig and the Container Components

When the component is a container, the cq:editConfig node of the container is used to populate configuration settings to the child components. For example, configuration settings for the edit bar buttons, control set layout (editbars, rollover), dialog layout (inline, floating) can be defined on the parent component and propagated to the child components that do not define their own **cq:editConfig**

Sling Models

A Sling Model is an annotation-driven java class that is automatically mapped to Sling objects, for example: Resource or Request. Sling models allow you to directly access JCR property values. A Sling model is implemented in an OSGi bundle.

You will be creating proxy components that extend the Core components. The business logic for the Core components is implemented in Sling Models. In order to meet project-specific requirements, it may be necessary to also customize the business logic for the Core component that is being extended. When extending the business logic for a Core component, use the Delegation Pattern for Sling Models.

Note: Sling Models will be covered in more detail later in this course.

Extended Component Use Case: Hero

The Hero component is an image with embedded title and call-to-action link.

Functional Requirements:

1. Must conform to site design look-and-feel (see wireframes from creative group)
2. Component design must match wireframes created by the creative group
3. Component must follow extensibility patterns of Adobe's Core Components
4. Author input:

Label	Datatype	Notes
Asset	Image from DAM	Must support drag-and-drop of image directly on to the page
Hero Title	String	Title of image
Get Title from DAM	Checkbox	default=false
Display Title as Popup	Checkbox	default=false
Link	pathBrowser	call to action link to a page
Link Text	String	descriptive text

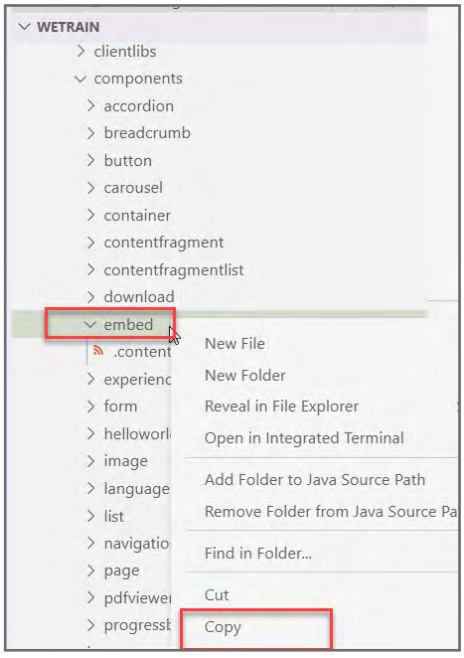
Exercise 1: Extend a Core Component

To use a hero image at the top of page mock-ups from the creative team, you must define and implement the hero image to be used on the pages. You start by extending a Core Component.
In this exercise you will extend a Core Component, then deploy and verify.
Prerequisites:

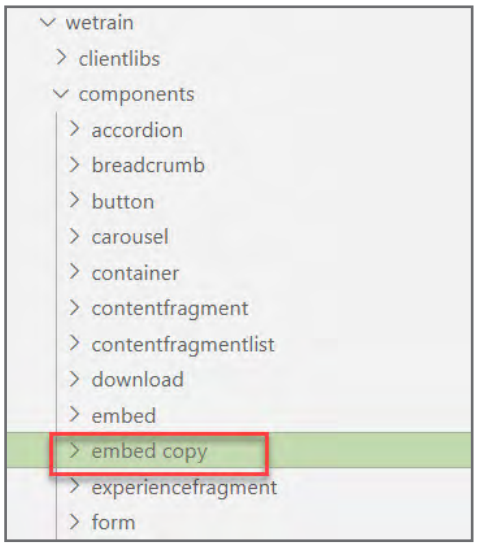
- Local author service running
- A Maven project imported into your IDE and installed on the author service

Task 1: Extend a Core Component

1. In your IDE, navigate to <A**EM Project>/ui.apps/src/main/content/jcr_root/apps/wetrain/components**.
2. Right-click the **embed** folder and select **Copy**, as shown:



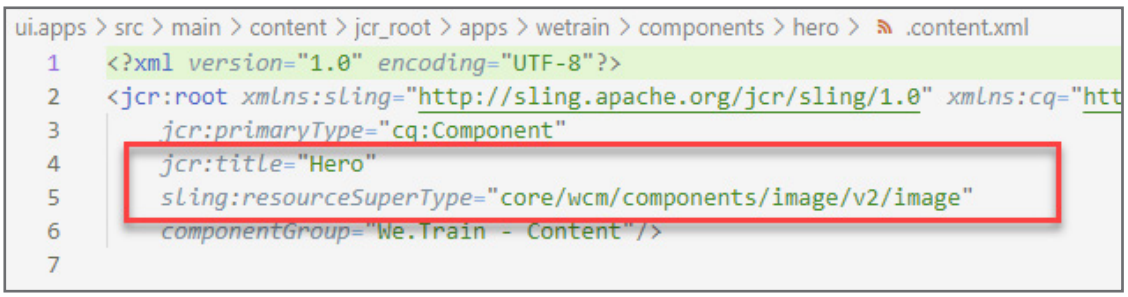
3. Paste the copied **embed** component folder under the **components** folder:



4. Rename the copied folder **hero**.
5. Expand the **hero** component folder and open the **.content.xml** file.
6. Set the following property values in the .content.xml file:

Name	Type	Value
jcr:title	String	Hero
sling:resourceSuperType	String	core/wcm/components/image/v2/image

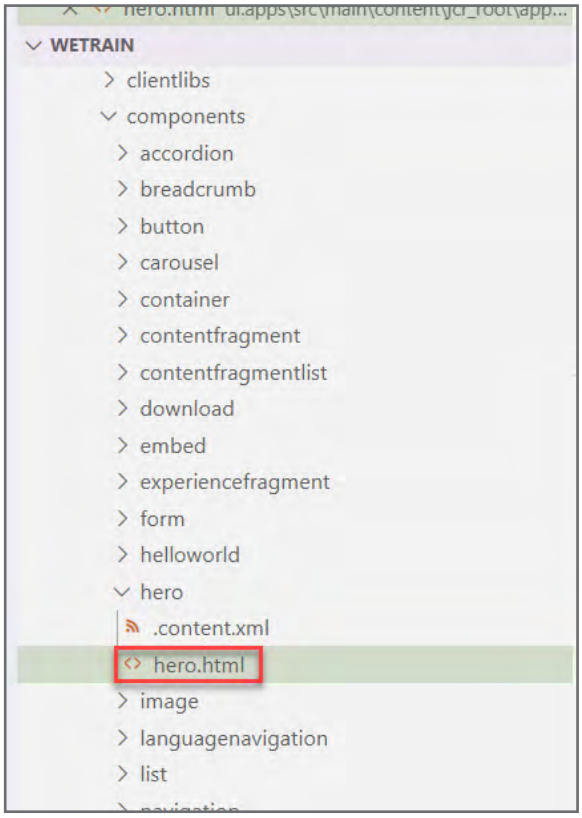
Setting the sling:resourceSuperType property has declared V2 of the core Image component to be the supertype of the Hero component.



7. Save the **.content.xml** file.

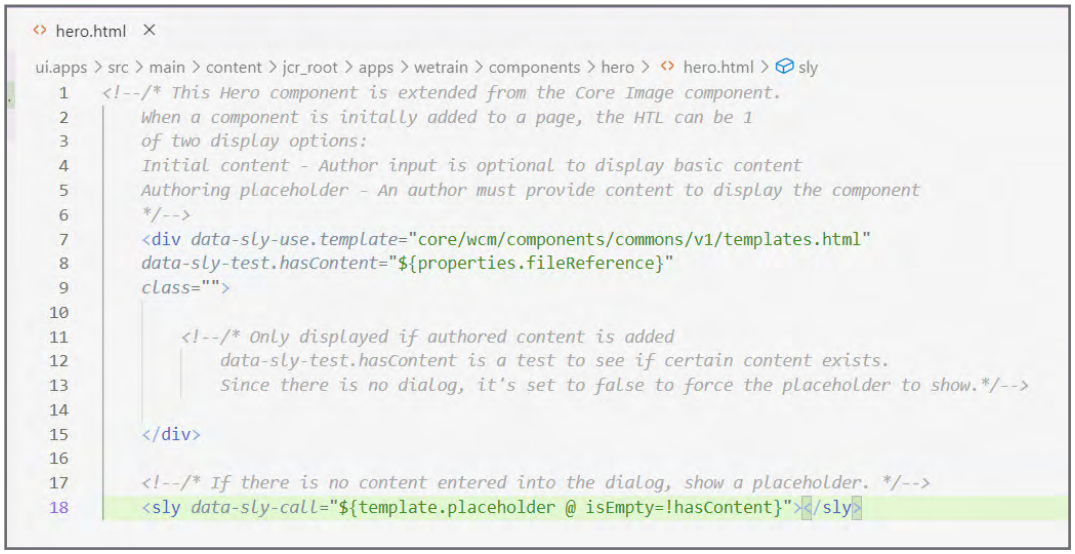
Note: When you save the file, the IDE exports the changes to the JCR repository.

8. Using your IDE, create a file named **hero.html** in the **hero** folder, as shown:



9. In the **Exercise_Files-DWC** folder, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/hero**.

10. Copy contents of the **initial_hero.html** file and paste it into the **hero.html** file that you just created.



11. Save the **hero.html** file (**File** > **Save**).

12. Examine the script. Notice the logic to display the placeholder if the component has no content.

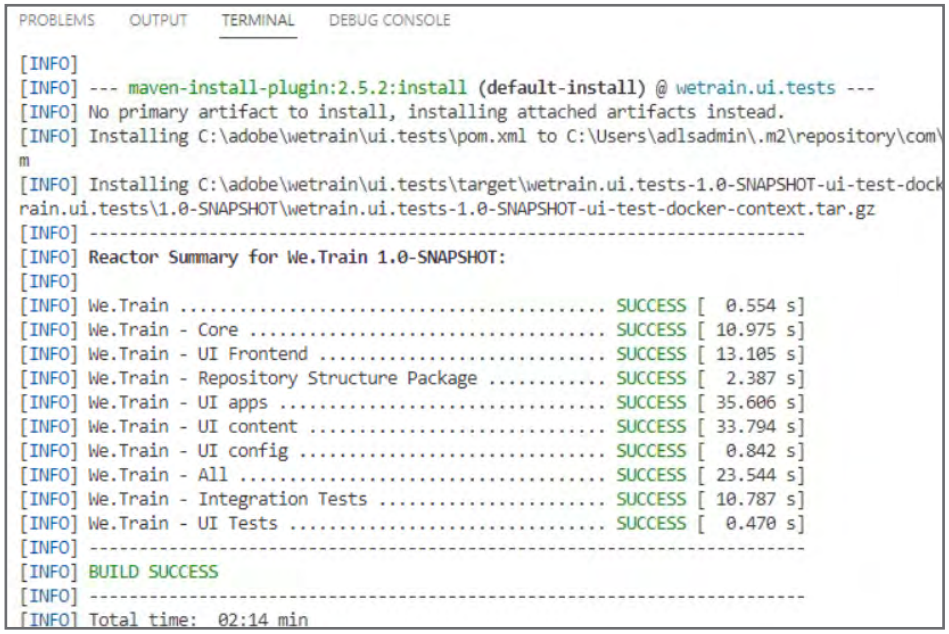
Task 2: Deploy and Verify

1. Using your IDE, open a terminal window.

Note: As an alternative to the IDE terminal window you could open an OS terminal window in your <AEM Project> folder.

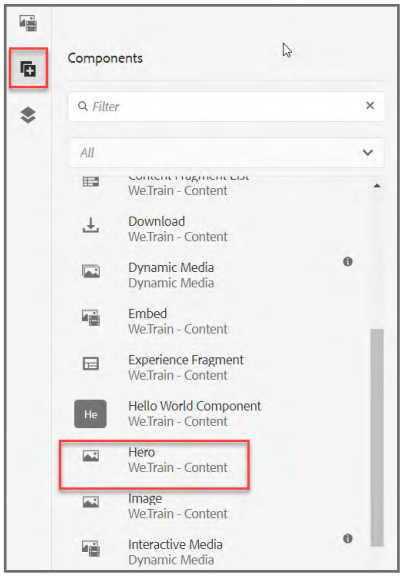
2. Type in the following command to deploy your project to the author service:

```
mvn clean install -PautoInstallSinglePackage
```



3. To verify the project deployment, use a browser to navigate to **Sites > We.Train > Language Masters.**

4. Open the **en** page in the page editor and expand the left rail.
5. Select the **Components** tab in the left rail. You will notice the **Hero** component is in the list of components.



6. Drag the **Hero component** onto the page, as shown.

Note: You see the Hero component in the left rail Component tab, even though it does not have a dialog box or cq:editConfig node. The Hero component appears in the Component tab because it is inheriting that functionality from the supertype.

Exercise 2: Customize the Component Dialog

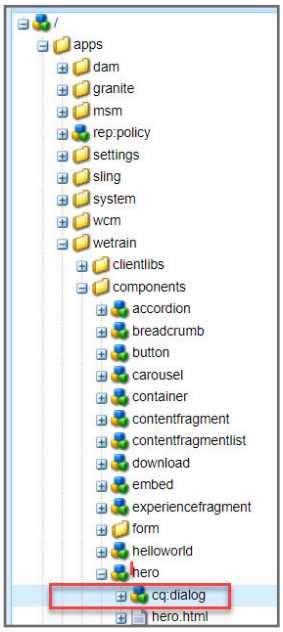
The Hero component is a proxy component for the core Image component and inherits its dialog box from that Image component. To make changes to the Hero component functionality, you need to add a custom dialog.
In this exercise you will create a custom dialog, import the changes to the Maven project, update the Hero component script, then deploy and verify.
Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service
- Hero component that is a proxy component for the core Image component

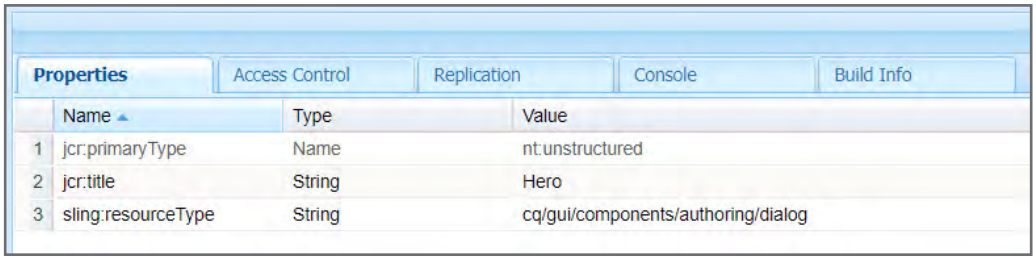
Task 1: Create a Custom Dialog

1. Using **CRXDE Lite**, navigate to the core Image component: **[/libs | /apps]/core/wcm/components/image/v2/image**.
2. Copy the **cq:dialog** node.

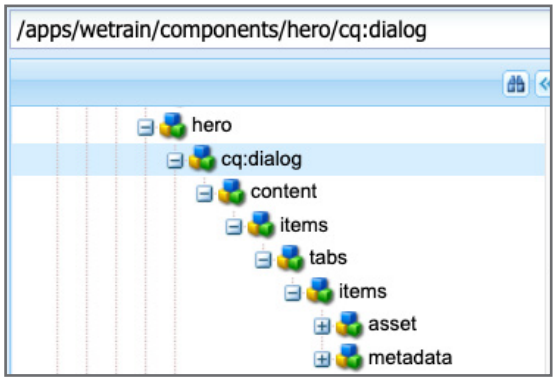
3. Navigate to **/apps/wetrain/components.** Paste the copied **cq:dialog** node under the **hero** component node, as shown:



4. Update the pasted **cq:dialog** node properties as shown:
   a. Delete **helpPath**
   b. Delete **extraClientlibs**
   c. Delete **trackingFeature**
   d. Update jcr:title = **Hero**

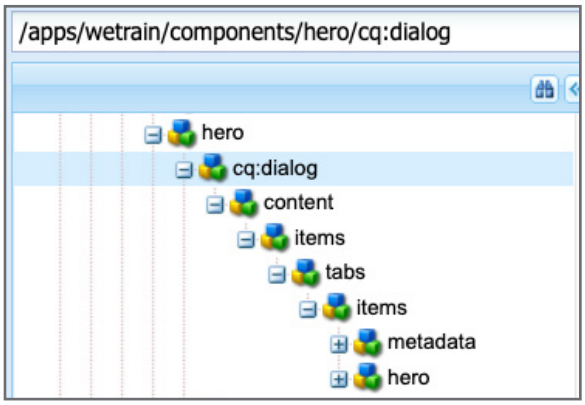


5. Expand the **cq:dialog** node.



Now delete the tabs that you want to inherit from the core Image component and create the custom tab that is needed to meet the requirements.

6. Delete the **asset** node and save the changes.
7. Right-click the **metadata** node and select **Copy** from the context menu.
8. Right-click the **items** node above the **metadata** node and select **Paste** from the context menu.
9. Rename the **metadata copy** node to **hero** and Save.

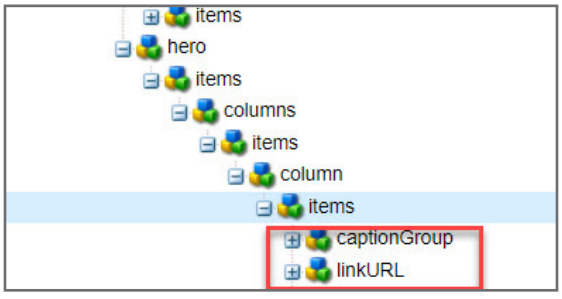


10. On the **hero** tab node, update the **jcr:title** property as shown:

Name	Type	Value
jcr:title	String	Hero

11. Expand the **hero** tab. Under **hero/items/columns/items/column/items** node, delete the following child nodes:
    a. dynamicmediaGroup
    b. decorative
    c. alternativeGroup
    d. id

    At this point, only the caption group and linkURL child nodes should remain, as shown:



12. Save the changes.

To modify the **hero** tab of the dialog box.

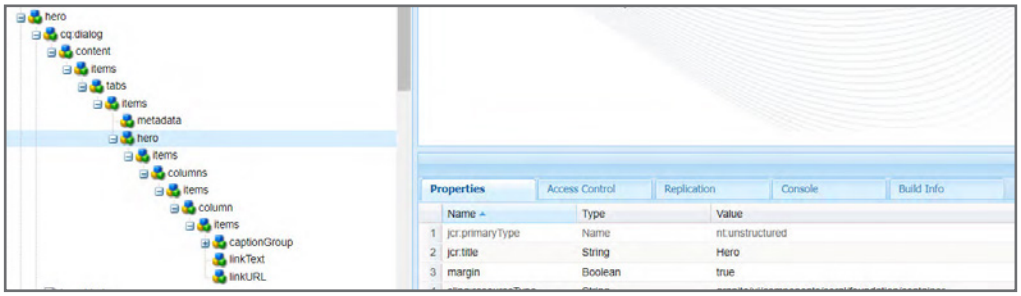
13. Expand the **captionGroup** node.
14. Right-click the **caption** node and rename to **heroTitle.**
15. Select the **heroTitle** node and modify the properties as shown:

Name	Type	Value
fieldLabel	String	Hero Title

16. Save the changes.
17. Right-click the **hero/items/columns/items/column/items** node and select **Create** > **Create Node** to create a node with the following values:
    - Name: **linkText**
    - Node Type: **nt:unstructured**
18. Add the following properties to the **linkText** node:

Name	Type	Value
sling:resourceType	String	granite/ui/components/coral/foundation/form/textfield
fieldLabel	String	Link Text
name	String	./linkText

19. Save the changes.
20. Rearrange the form field nodes in the order **captionGroup**, **linkText**, and then **linkURL** so that the dialog will display the fields in the desired order.



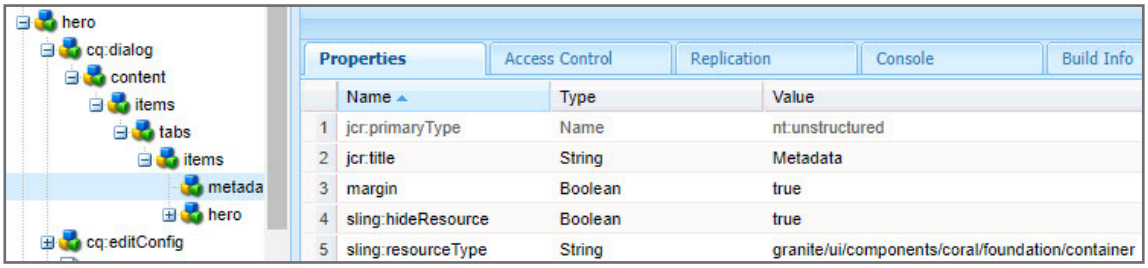
21. Save the changes.

Since the requirements call for a simplified hero, you do not want to show the metadata tab.

22. On the **metadata** tab node, add a property, as shown:

Name	Type	Value
sling:hideResource	Boolean	true

23. Delete the **metadata/items** node.



24. Save the changes.

Task 2: Import the Changes to the Maven Project

You have made the Hero component dialog changes directly in the JCR repository, using **CRXDE Lite**. You must now export the changes from the JCR repository and import them into your Maven project so that the changes will be retained when you next deploy your project.

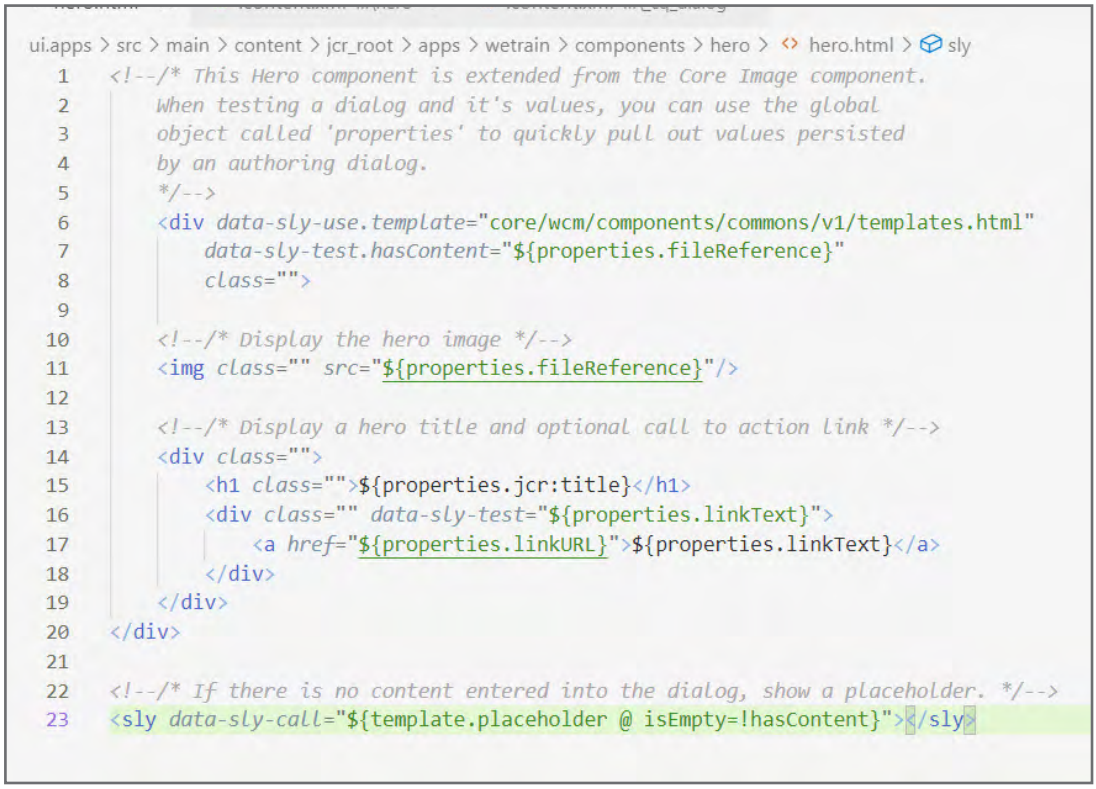
1. Using your IDE, right-click on the **hero** component folder and select **Import from AEM Server.**
2. Expand the **hero** component folder. You see a **cq:dialog** folder.
3. Expand the **cq:dialog** folder and open the **.content.xml** file. You see the dialog nodes and properties that you defined.

ui.apps > src > main > content > jcr_root > apps > wetrain > components > hero > _cq_dialog > .content.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:sling="http://sling.apache.org/jcr/sling/1.0" xmlns:granite="http://www.adobe.com/jcr/granite/1.0" xmlns:cq
    jcr:primaryType="nt:unstructured"
    jcr:title="Hero"
    sling:resourceType="cq/gui/components/authoring/dialog">
    <content
        granite:class="cmp-image__editor"
        jcr:primaryType="nt:unstructured"
        sling:resourceType="granite/ui/components/coral/foundation/container">
        <items jcr:primaryType="nt:unstructured">
            <tabs
                jcr:primaryType="nt:unstructured"
                sling:resourceType="granite/ui/components/coral/foundation/tabs"
                maximized="{Boolean}true">
                <items jcr:primaryType="nt:unstructured">
                    <metadata
                        jcr:primaryType="nt:unstructured"
                        jcr:title="Metadata"
                        sling:hideResource="true"
                        sling:resourceType="granite/ui/components/coral/foundation/container"
                        margin="{Boolean}true"/>
                    <hero
                        jcr:primaryType="nt:unstructured"
                        jcr:title="Hero"
                        sling:resourceType="granite/ui/components/coral/foundation/container"
                        margin="{Boolean}true">
                        <items jcr:primaryType="nt:unstructured">
                            <columns
                                jcr:primaryType="nt:unstructured"
                                sling:resourceType="granite/ui/components/coral/foundation/fixedcolumns"
                                margin="{Boolean}true">
                                <items jcr:primaryType="nt:unstructured">
```

Task 3: Update the Hero Component Script

1. In the Exercise_Files-DWC folder, navigate to **ui.apps/src/main/content/jcr_root/apps/components/hero**.
2. Copy contents of the **dialog_hero.html** file and paste it to replace the contents of the **hero.html** file in your IDE.
3. Examine the script to see that the script renders properties that are written by the dialog box. Notice also that the fileReference property is written by the inherited assets tab of the dialog box.



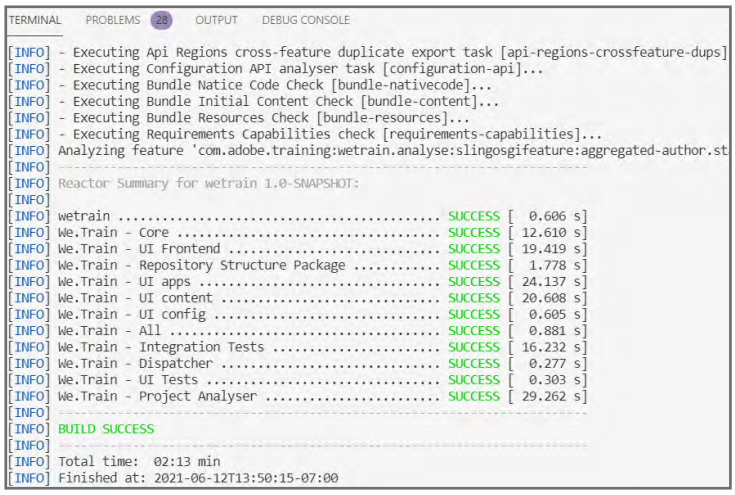
Task 4: Deploy and Verify

1. Using your IDE, open a terminal window.

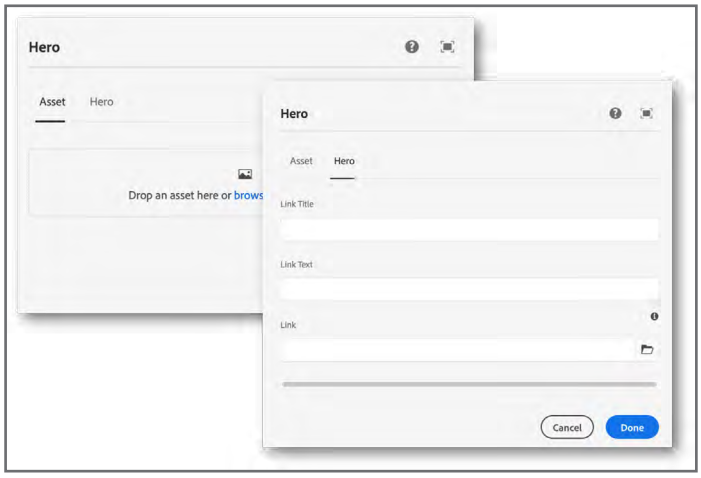
Note: As an alternative to the IDE terminal window you could open an OS terminal window in your <AEM Project> folder.

2. Type in the following command to deploy your project to the author service:

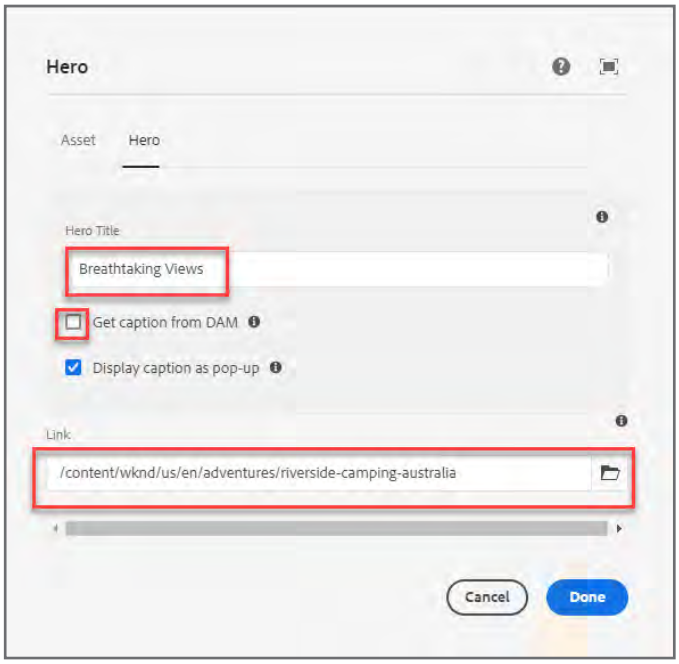
```
mvn clean install -PautoInstallSinglePackage
```



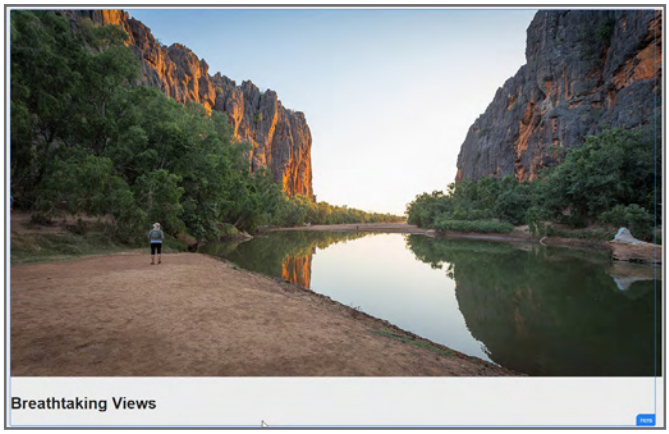
3. To verify the project deployment, use a browser to navigate to **Sites** > **We.Train** > **Language Masters**.
4. Open the **en** page in the page editor.
5. If no Hero component is on the page, drag a **Hero component** onto the page.
6. Open the left rail and select the **Asset** tab.
7. Click on the Hero placeholder to open the component toolbar.
8. Click Configure (wrench) to open the dialog box.



9. In the left rail, type **camping** into the asset browser filter.
10. Drag an image into the Asset tab of the dialog.
11. Click the Hero tab and enter **Breathtaking Views** into the Hero Title.
12. Uncheck **Get Caption from DAM**.
13. Click on the path browser to the right of the **Link** field and navigate to **WKND Site > United States > WKND Adventures and Travel > Adventures > Colorado Rock Climbing.**



14. Click **Done**.



Exercise 3: Enable Authoring Actions Using cq:editConfig

The business requirements for the Hero component specify that the author should be able to drag-and-drop an image directly onto the component without opening the configure dialog. As a result, the developer must add a cq:editConfig definition to the Hero component.
In this exercise you will enable drag-and-drop for the Hero component, import the changes to the Maven project, then test the editConfig drag and drop.

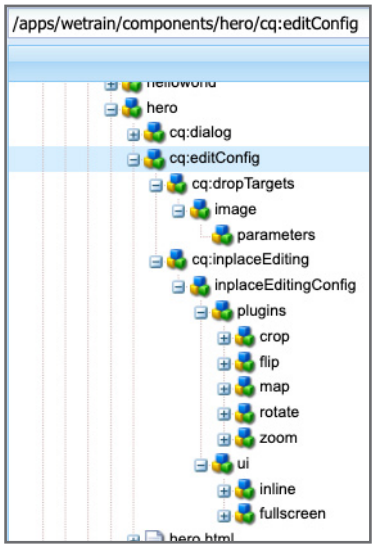
Prerequisites:

- Local author service running
- A maven project imported into your IDE and installed on the author service
- Hero component extending the image component with an extended dialog

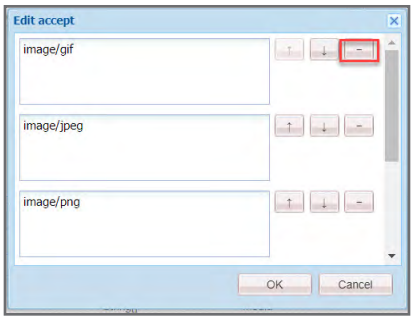
Task 1: Enable Drag-and-Drop for the Hero Component

1. Using CRXDE Lite, navigate to the core Image component: **[/libs | /apps]/core/wcm/components/image/v2/image**.
2. Copy the **cq:editConfig** node.
3. Navigate to /**apps/wetrain/components** and paste the copied **cq:editConfig** node under the **hero** component node.
4. Save the changes.

5. Expand the **cq:editConfig** node and examine the child node structure and properties.



6. On the **dropTargets/image** node, modify the **accept** property so that only **image/jpeg** remains.
   - Double-click the Property Value field for accept. The Edit accept dialog opens.
   - Delete the values **image/gif**, **image/png**, **image/tiff, image/svg\+xml** by clicking the - button, as shown:



Keep only **image/jpeg,** as shown:

Name	Type	Value
accept	String	image/jpeg

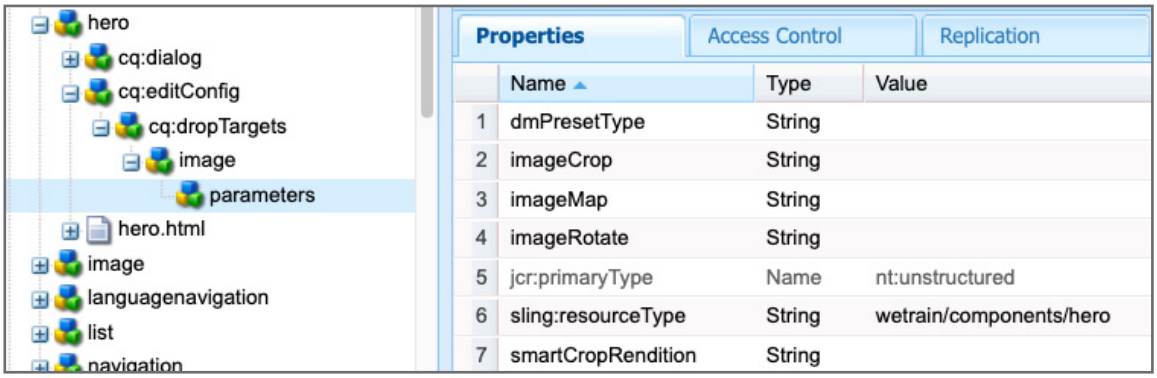
7. Save the changes.

8. On the **dropTargets/image/parameters** node, add a **sling:resourceType** property as shown and Save.

Name	Type	Value
sling:resourceType	String	wetrain/components/hero

9. Save the changes.

10. Under the **cq:editConfig** node, delete the **cq:inplaceEditing** node and Save.
11. Verify the changes made.



Task 2: Import the Changes to the Maven Project

You have made the Hero component changes, directly in the JCR repository, using CRXDE Lite. You must now export the changes from the JCR repository and import them into your Maven project so that the changes will be retained when you next deploy your project.

1. Using your IDE, right-click on the hero component folder and select **Import from Server**.
2. Expand the **hero** component folder. You see a **cq:editConfig.xml** file.
3. Open the **_cq:editConfig.xml**. You see described the editConfig nodes and properties that you defined.

ui.apps > src > main > content > jcr_root > apps > wetrain > components > hero > _cq_editConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:sling="http://sling.apache.org/jcr/sling/1.0" xmlns:cq="h
    jcr:primaryType="cq:EditConfig">
    <cq:dropTargets jcr:primaryType="nt:unstructured">
        <image
            jcr:primaryType="cq:DropTargetConfig"
            accept="[image/jpeg]"
            groups="[media]"
            propertyName="./fileReference">
            <parameters
                jcr:primaryType="nt:unstructured"
                sling:resourceType="wetrain/components/hero"
                imageCrop=""
                imageMap=""
                imageRotate=""/>
        </image>
    </cq:dropTargets>
</jcr:root>
```

Task 3: Test the editConfig Drag and Drop

1. To verify the cq:editConfig node structure, use a browser to navigate to **Sites > We.Train > Language Masters**.
2. Open the **en** page in the page editor.
3. Open the left rail.
4. If a Hero component does not exist on the page, drag a Hero component onto the page.
5. Click the **Assets** tab in the left rail.
6. Type **png** into the search filter to find all the images of mime type **png**.
7. Attempt to drag an image and drop it on to the Hero component. The Hero component should not accept the images of mime type **png**.
8. Type **jpeg** into the search filter to find all the images of mime type **jpeg**.
9. Drag an image and drop it on to the Hero component. Examine what happens and the reason for it.

Exercise 4: Add a Default Design to a Component

For a newly created proxy component to conform to the website design standards, you must add the component-specific client library files for the component.
In this exercise you will add BEM classes to the component script, add the component specific client library files, then deploy and verify.
Prerequisites

- Local author service running
- A Maven project imported into your IDE and installed on the author service
- Hero component that is a proxy component for the core Image component

Task 1: Add BEM classes to the Component Script

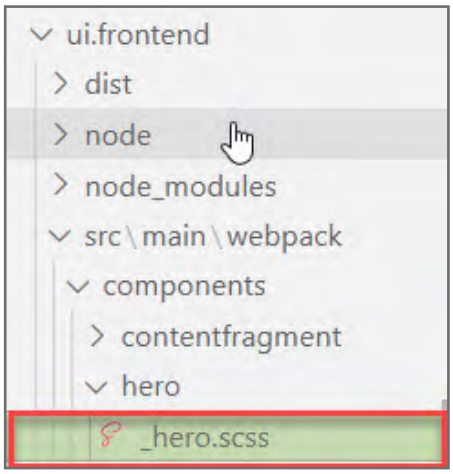
1. In the Exercise_Files-DWC folder, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/hero**.
2. Copy contents of the **design_hero.html** file.
3. Using your IDE, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/hero** and paste to replace the contents of the **hero.html** file in your Maven project.

4. Examine the script and you will see that the script makes use of the CSS classes to render the image and the styled **jcr:title** and **linkText** properties:

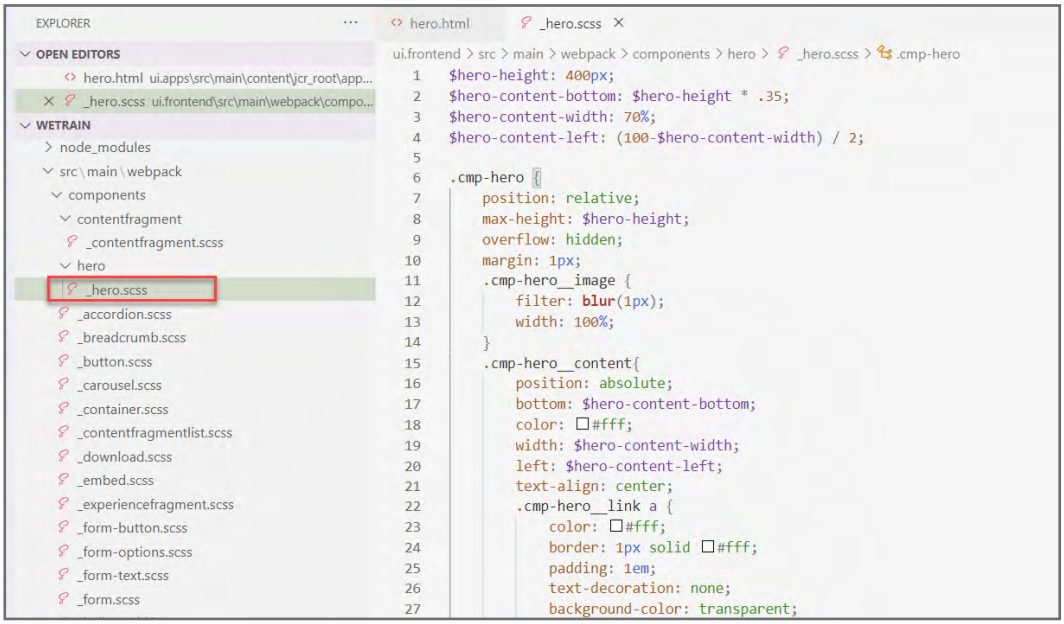
```
<!--/* This Hero component is extended from the Core Image component.
    */-->
    <div data-sly-use.hero="com.adobe.training.core.models.Hero"
        data-sly-use.template="core/wcm/components/commons/v1/templates.html"
        data-sly-test.hasContent="${!hero.empty}"
        class="cmp-hero">

    <!--/* Display the hero image */-->
    <img class="cmp-hero__image" alt="${hero.alt}" src="${hero.src}"/>

    <!--/* Display a hero title and optional call to action link */-->
    <div class="cmp-hero__content">
        <h1 class="cmp-hero__title">${hero.title}</h1>
        <div class="cmp-hero__link" data-sly-test="${hero.linkText}">
            <a href="${hero.link}">${hero.linkText}</a>
        </div>
    </div>
</div>

<!--/* If there is no content entered into the dialog, show a placeholder. */-
<sly data-sly-call="${template.placeholder @ isEmpty=!hasContent, classAppend=
```

Task 2: Add the Component Specific Client Library Files

1. Using your IDE, navigate to **<AEM Project>/ui.frontend/src/main/webpack/components**.
2. Under the **components** folder, create a folder named **hero**.
3. Under the **hero** folder, create a file named **_hero.scss**. This will be the default design for the hero component.



4. In your Exercise_Files-DWC folder, navigate to **ui.frontend/src/main/webpack/components/hero** folder.
5. Copy contents of the **_hero.scss file** and paste it into the **_hero.scss** file that you just created.



6. Save the file.
7. Examine the design file and familiarize yourself with the class names and design elements.

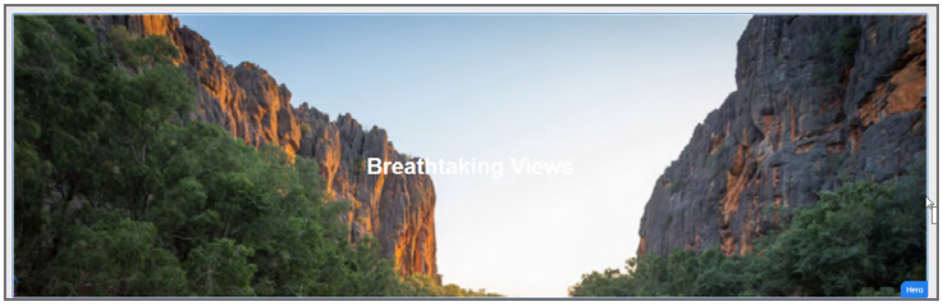
Task 3: Deploy and Verify

1. Using your IDE, open a terminal window.
2. Type in the following command to deploy your project to AEM:

```
mvn clean install -PautoInstallSinglePackage
```

```
TERMINAL   PROBLEMS 28   OUTPUT   DEBUG CONSOLE

[INFO] - Executing Api Regions cross-feature duplicate export task [api-regions-crossfeature-dups]
[INFO] - Executing Configuration API analyser task [configuration-api]...
[INFO] - Executing Bundle Natice Code Check [bundle-nativecode]...
[INFO] - Executing Bundle Initial Content Check [bundle-content]...
[INFO] - Executing Bundle Resources Check [bundle-resources]...
[INFO] - Executing Requirements Capabilities check [requirements-capabilities]...
[INFO] Analyzing feature 'com.adobe.training:wetrain.analyse:slingosgifeature:aggregated-author.st
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary for wetrain 1.0-SNAPSHOT:
[INFO]
[INFO] wetrain ..................................... SUCCESS [  0.606 s]
[INFO] We.Train - Core ............................. SUCCESS [ 12.610 s]
[INFO] We.Train - UI Frontend ...................... SUCCESS [ 19.419 s]
[INFO] We.Train - Repository Structure Package ..... SUCCESS [  1.778 s]
[INFO] We.Train - UI apps .......................... SUCCESS [ 24.137 s]
[INFO] We.Train - UI content ....................... SUCCESS [ 20.608 s]
[INFO] We.Train - UI config ........................ SUCCESS [  0.605 s]
[INFO] We.Train - All .............................. SUCCESS [  0.881 s]
[INFO] We.Train - Integration Tests ................ SUCCESS [ 16.232 s]
[INFO] We.Train - Dispatcher ....................... SUCCESS [  0.277 s]
[INFO] We.Train - UI Tests ......................... SUCCESS [  0.303 s]
[INFO] We.Train - Project Analyser ................. SUCCESS [ 29.262 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  02:13 min
[INFO] Finished at: 2021-06-12T13:50:15-07:00
```

3. To verify the project deployment, use a browser to navigate to **Sites** > **We.Train** > **Language Masters**.
4. Open the **en** page in the page editor.
5. If a Hero component does not exist on the page, drag a Hero component onto the page. Open the Configure Dialog, drag an image into the dialog and populate all the fields.
   You should now see the Hero component fully styled, as defined in the style sheet.



(Optional) Exercise 5: Create a Sling Model for Headless Output

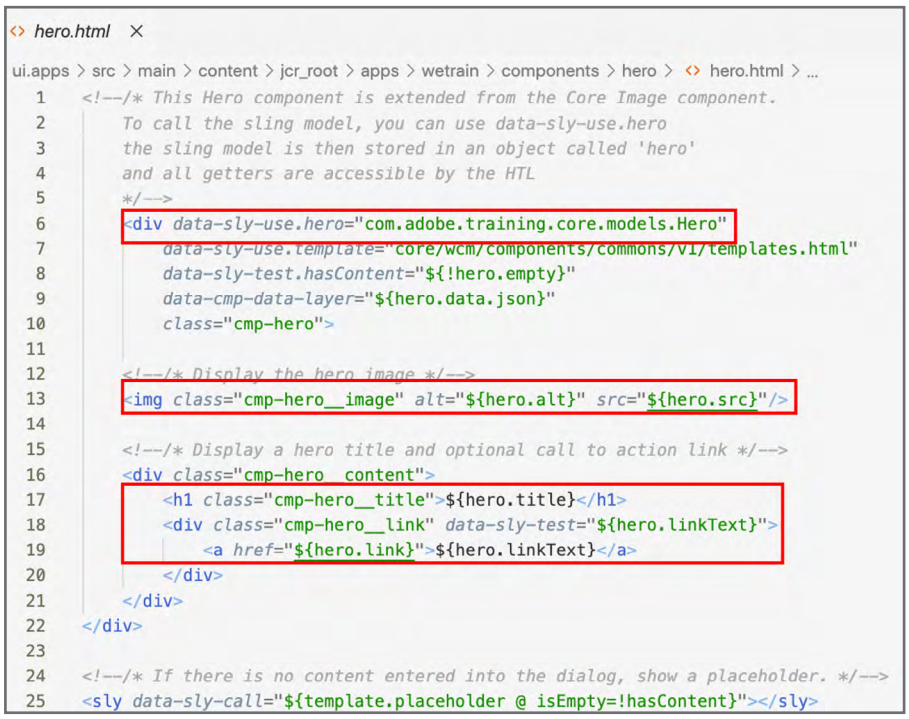
You can export AEM-managed content to an external delivery channel. To do this, you can take advantage of a component Sling Model to export component content in a safe, well documented JSON format.
In this exercise you will update the Hero component script, update the Sling Model, then deploy and verify.
Prerequisites

- Local author service running
- A maven project imported into your IDE and installed on the author service
- Hero component that is a proxy component for the core Image component

Task 1: Update the Hero Component Script

1. In the Exercise Files folder, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/hero**.
2. Copy contents of the **slingmodel_hero.html** file.
3. Using your IDE, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/hero** and paste to replace the contents of the **hero.html** file in your Maven project.

4. Examine the script and you will see that the script makes use of the Sling Model as a helper object. The Sling Model will return an object named "hero" which contains the Hero component property values.



Task 2: Update the Sling Model

1. In the file system, delete the **<AEM Project>/core/src** folder.
2. In the **Exercise_Files-DWC** folder, navigate to and copy the **core/src** folder.
3. Paste the **src** folder into your <AEM Project>. This will give you all the supporting back-end logic required.

Task 3: Deploy and Verify

1. Using your IDE, open a terminal window.
2. Type in the following command to deploy your project to the author service:

   ```
   mvn clean install -PautoInstallSinglePackage
   ```

3. To verify the project deployment, use a browser to navigate to **Sites** > **We.Train** > **Language Masters**.
4. Open the **en** page in the page editor.
5. If a Hero component does not exist on the page, drag a **Hero** component onto the page.
6. Open the Configure dialog, drag an image into the dialog, and populate all the fields.
7. Use the Page Information dropdown menu and click **View as Published** to open the page without the editor.
8. Modify the URL of the page by removing any extension and suffix. Then type **model.json** as the extension to see the json output.
   For example: http://localhost:4502/content/wetrain/us/en.model.json
9. Search for **"hero": {** on the page to find the Hero component output:

References

- Developing Core Components
- Paths to Success with Core Components
- Component Customization Patterns
- cq:editConfig Properties
- cq:editConfig Child Nodes
- Sling Models
- Customizing Core Component Logic
- Sling Model Delegation Pattern.

Implement Custom Components

Introduction

Adobe Experience Manager (AEM) comes with out-of-the-box components, but depending on your business needs a custom component might be necessary. If custom components are needed, a developer can still use the core components as good references for reusability and development patterns. In this module, you will learn how to create a custom component and many other component development features that can enhance your components to be modular, reusable, and upgrade safe.

Objectives

After completing this module, you will be able to:

- Explain the important features of components
- Create a custom component
- Add a dialog field validation to the custom component
- Add a default design to a component
- Add a content policy style tab to a component
- Use the Sling model exporter for a component
- Create localization information

Features of Components

Some common component features are:

- **Component Placeholder**: Helps validate if the component is on the page and is available only on the page Edit mode. The placeholder is not available on the page Preview mode or on the publish service. If the content is added to the component, the placeholder does not appear on the page.
- **Dialog Validation**: Provides a straightforward validation framework that helps create custom form element validators and interfaces with them programmatically. Registering custom validators is done by calling a jQuery based `$.validator.register` method. The register method takes a single JavaScript object literal argument. The parameter looks for four properties: `selector`, `validate`, `show`, and `clear`, of which only selector is required.
- **Style System**: Enables a template author to define style classes in the content policy of a component so that a content author can select them when editing the component on a page. These styles provide alternative visual variations of a component. This eliminates the need to develop a custom component for each style or to customize the component dialog to enable such style functionality. It leads to more reusable components that can be quickly and easily adapted to the needs of content authors without any AEM back-end development.
- **Sling Models for Components**: When developing an AEM project, you can define a model object (a Java object) and map that object to Sling resources. A Sling model is implemented as an OSGi bundle. A Java class located in the OSGi bundle is annotated with `@Model` and the adaptable class (for example, `@Model(adaptables = Resource.class)`. The data members (Fields) use `@Inject` annotations. These data members map to node properties.
- **Sling Model Exporter**: Enables new annotations to be added to Sling Models that define how the Model can be exported as JSON. With Sling Model Exporter, you can obtain the same properties as a JSON response, without creating a Sling Servlet. You need to export the Sling Model by using the Jackson exporter. The Sling Model Exporter can be used as a web service or as a REST API.
- **Selectors**: Provides a way to choose the script to be rendered when a user requests a page. During the resource resolution step, if the first character in the request URL after the resource path is a dot (.), the string after the dot up to (but not including the last dot before the slash character, or the end of the request URL) comprises the selectors. If the resource path spans the complete request URL, no selectors exist. Also, if only one dot follows the resource path before the end of the request URL or the next slash, no selectors exist.

- **i18n Translation**: Provides a Strings & Translations console for managing the various translations of texts used in the component UI. The translator tool helps manage English strings and their translations. The dictionaries are created in the repository. From this console, you can search, filter, and edit both English and translated texts. You can also export dictionaries to XLIFF format for translating, and then import the translations back into the dictionaries. It is also possible to add the i18n dictionaries to a translation project from the console. You can either create a new dictionary or add a dictionary to an existing project.

Custom Component Use Case: Stockplex

The Stockplex component takes in a stock symbol and displays information about the recent history of the stock.

Functional Requirements:

1. Must conform to site design look-and-feel (see wireframes from creative group).
2. Component design must match wireframes created by the creative group.
3. Author input:

Label	Datatype	Notes
Stock Symbol	String	Must be 4 letter alpha only
Summary of stock	String	Description of stock

4. Template Configuration to optionally show detailed stock information when the stockplex component is added to a page.
5. As shown in the wireframes, the author should be able to select styling from a bound list that changes the components text based on the component style sheets.
6. The We.Train website is a multinational, multilingual site. Strings generated by code must be translated into all required languages.
7. Component must export rendered content in both fully formatted HTML and secure, well documented JSON formats.
8. Component must export relevant user experience data into a data layer, accessible by Adobe Analytics and Adobe Target.

Exercise 1: Create a Custom Component

The first step to create a custom component is to get something to render onto the page. This will allow you to develop user author inputs and test outputs in later exercises.

Prerequisites:

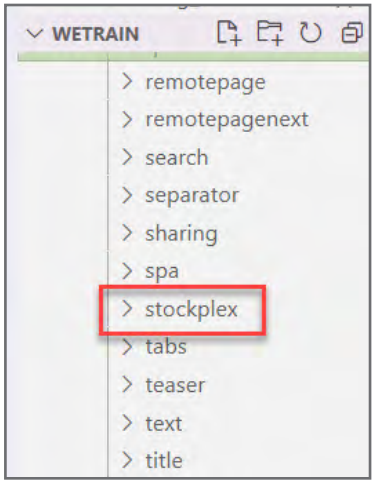
- Local author service running
- A Maven project imported into your IDE and installed on the author service

This exercise includes the following tasks:

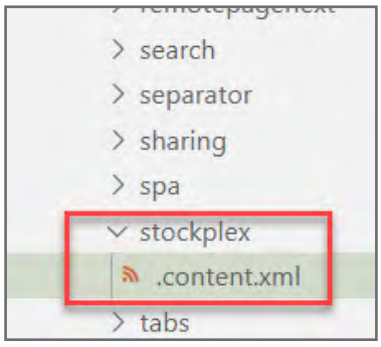
1. Create a Custom Component
2. Deploy and Verify

Task 1: Create a Custom Component:

1. In your IDE, navigate to **<AEM Project>/ui.apps/src/main/content/jcr_root/apps/wetrain/components**.
2. Copy the **embed** component folder.
3. Paste the **copied embed component folder** under the **components** folder.
4. Rename the copied folder to **stockplex**:



5. Expand the **stockplex** component folder and open the **.content.xml file**, as shown:



6. Delete the **sling:resourceSuperType** property from the **.content.xml** file.
7. Modify the **jcr:title** property and add the **jcr:description** property, as shown:

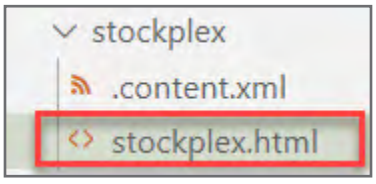
Name	Type	Value
jcr:title	String	Stockplex
jcr:description	String	We.Train complex stock component

```
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:sling="http://sling.apache.org/jcr/sling
    jcr:primaryType="cq:Component"
    jcr:title="Stockplex"
    jcr:description="We.Train complex stock component"
    componentGroup="We.Train - Content"/>
```

8. **Save** the .content.xml file.

Note: When you save the file, the IDE exports the changes to the AEM repository.

9. In your IDE, create a file named **stockplex.html** in the **stockplex** folder, as shown:



10. In the Exercise_Files-DWC folder, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/stockplex.**

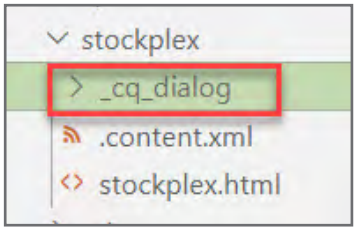
11. **Copy** contents of the **initial_stockplex.html** file and **paste** it into the **stockplex.html** file that you just created:

```
<!--/* When a component is initally added to a page, the HTL can be 1
    of two display options:
    Initial content - Author input is optional to display basic content
    Authoring placeholder - An author must provide content to display the compone
    */-->
    <div data-sly-use.template="core/wcm/components/commons/v1/templates.html"
    data-sly-test.hasContent="${properties.symbol}"
    class="">

    <!--/* Only displayed if authored content is added
        data-sly-test.hasContent is a test to see if certain content exists.
        Since there is no dialog, it's set to false to force the placeholder to s

</div>

<!-- If there is no stock symbol added to the dialog, create a component placehol
<sly data-sly-call="${template.placeholder @ isEmpty=!hasContent}"></sly>
```

12. Save the changes.

You must create a **cq:dialog** node to enable the component in the authoring UI.

13. In your IDE, create a folder, under the **stockplex** component folder, named **_cq_dialog**.



14. Copy the **.content.xml** file from the parent **stockplex** folder into the **_cq_dialog** folder.
15. Edit the **.content.xml** file in the **_cq_dialog** folder. Delete the attributes **jcr:title**, **jcr:description**, and **componentGroup**. Only the **jcr:primaryType** should remain.
16. Modify the node type value so that **jcr:primaryType="nt:unstructured"**, as shown:

17. Verify the file ends with **/>**, as shown in the screenshot above.
18. Save the changes.

Task 2: Deploy and Verify

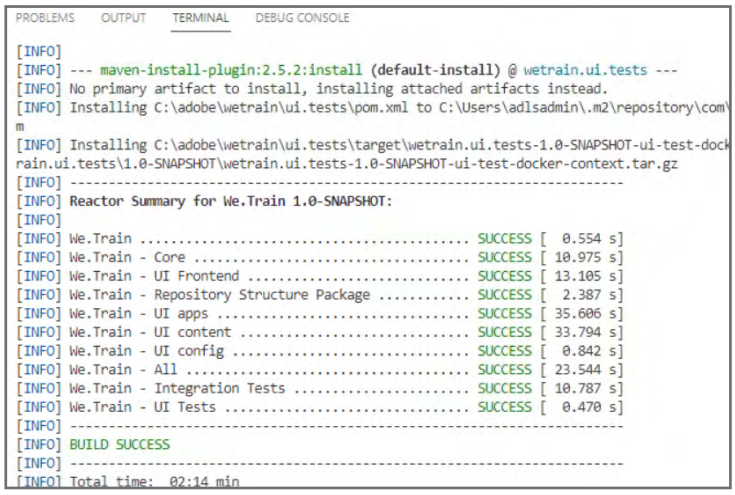
The IDE should have synched any saved changes to the project. However, we execute this build to be sure that all changes are synched from the project to the repository.

1. In your IDE, open a terminal window.

Note: As an alternative to the IDE terminal window you could open an OS terminal window in your <AEM Project> folder.

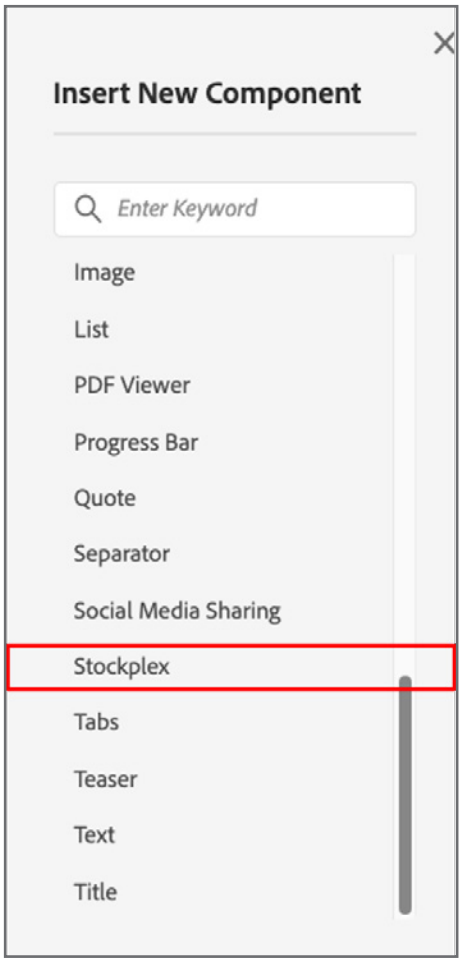
2. Type in the following command to deploy your project to AEM:

```
mvn clean install -PautoInstallSinglePackage
```

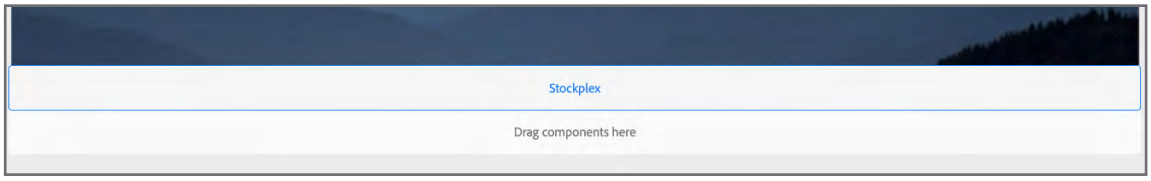


3. To verify the project deployment, use a browser to navigate to **Sites** > **We.Train** > **Language Masters**.
4. Open the **en** page in the page editor.

5. On the page, click the **Drag components here** placeholder and click the Insert component icon (+ icon) from the component toolbar. The Insert New Component dialog box opens.



6. Select the **Stockplex** component from the list. The stockplex component is added to the page, as shown:



Exercise 2: Create a Custom Dialog with Validation

The requirements document for a stockplex component specifies that the component accept only 4-letter stock symbols. You must add validation to the stockplex dialog to meet this requirement.
In this exercise you will create a dialog, update the stockplex script and verify the changes to the dialog. You will also add symbol validation, verify the dialog validation, then import the changes to your AEM project.

Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service
- Existing stockplex component with stub dialog

Task 1: Create the Dialog

1. In your IDE, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/stockplex/_cq_dialog**.
2. Open the file **.content.xml**.
3. In your Exercise_Files-DWC, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/stockplex/_cq_dialog**.
4. Open the **.content.xml** file and copy the contents.

5. Paste the copied contents in **.content.xml** file in your IDE.

6. Save the file.

Remember, when you saved the file, the IDE AEM plugin automatically synched back to the repository.

7. Scroll down in the **.content.xml** file to **content/items/tabs/items** properties, as shown:

This is the **Properties tab** in the dialog.

8. Continue to scroll down the properties tab and notice there are two form fields on the Properties tab: **symbol** and **summary.**

```
<column
    jcr:primaryType="nt:unstructured"
    sling:resourceType="granite/ui/components/coral/foundation/container">
    <items jcr:primaryType="nt:unstructured">
        <symbol
            jcr:primaryType="nt:unstructured"
            sling:resourceType="granite/ui/components/coral/foundation/form/textfield"
            fieldDescription="Enter a 4 character stock symbol"
            fieldLabel="Stock Symbol"
            name="./symbol"/>
        <summary
            jcr:primaryType="nt:unstructured"
            sling:resourceType="granite/ui/components/coral/foundation/form/textfield"
            fieldDescription="Enter a summary description of the stock"
            fieldLabel="Summary of Stock"
            name="./summary"/>
    </items>
```

Task 2: Update the Stockplex Script

1. In the Exercise_Files-DWC folder, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/stockplex.**
2. Copy contents of the **author_stockplex.html** file and **paste** it into the **stockplex.html** file in your IDE, as shown:

ui.apps > src > main > content > jcr_root > apps > wetrain > components > stockplex > stockplex.html > sly

```
<!--/* When authoring a component using a dialog, you can use the global
    object called 'properties' to quickly pull out values persisted to the JCR.
    */-->
    <div data-sly-use.template="core/wcm/components/commons/v1/templates.html"
    data-sly-test.hasContent="${properties.symbol}"
    class="">

    <div class="">${properties.symbol}</div>
    <div class="">Current Value: 300</div>

    <div data-sly-test.summary="${properties.summary}">
        <h3>Summary: ${properties.summary}</h3>
    </div>

    <div class="">
        <a href="#">
            <button>Placeholder</button>
        </a>
    </div>

    <ul data-sly-list.sInfo="${['Request Date','Open Price', 'Range High', 'Rang
        <li class="">
            <span class="">${sInfo}: value</span>
        </li>
    </ul>
</div>
```

3. Save the changes (**File** > **Save**).

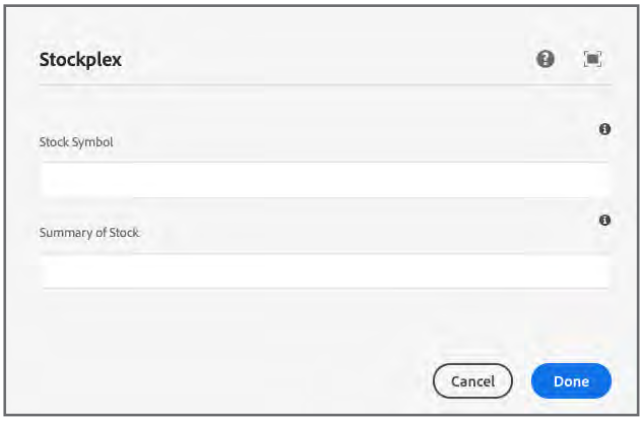
4. Examine the script to see how the Stockplex component renders the properties.

Note: If your IDE has auto-sync on save enabled, the .content.xml and stockplex.html file are already imported into AEM. If you are not using this feature, you will need to run maven install:

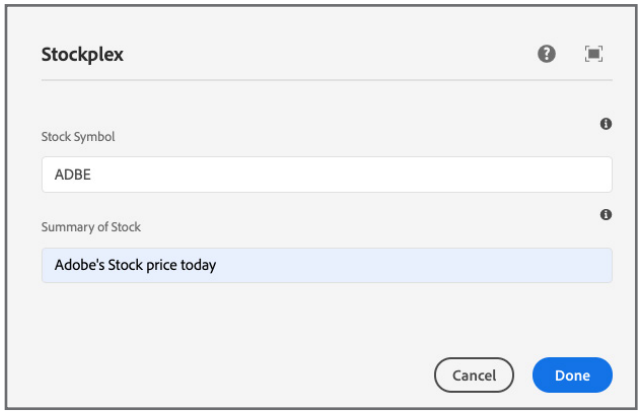
```
mvn clean install -Padobe-public,autoInstallSinglePackage
```

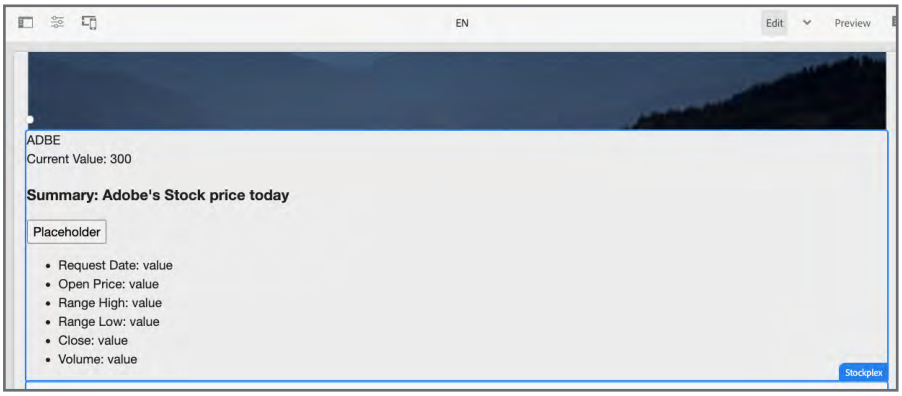
Task 3: Verify the Changes to the Dialog

1. In your browser, open the AEM homepage and navigate to **Sites** > **We.Train** > **Language Masters.**
2. Open the **en** page in the page editor.
3. If the **stockplex** component is not already on the page, drag one onto the page.
4. Click the **stockplex** placeholder to open the Component toolbar.
5. Click the **wrench** to open the stockplex configure dialog. You should see the two form fields: **Stock Symbol** and **Summary of Stock.**



6. Type the following values into the dialog and click **Done** to save the values.
   a. Stock Symbol**: ADBE**
   b. Summary of Stock: **Adobe's stock price today**





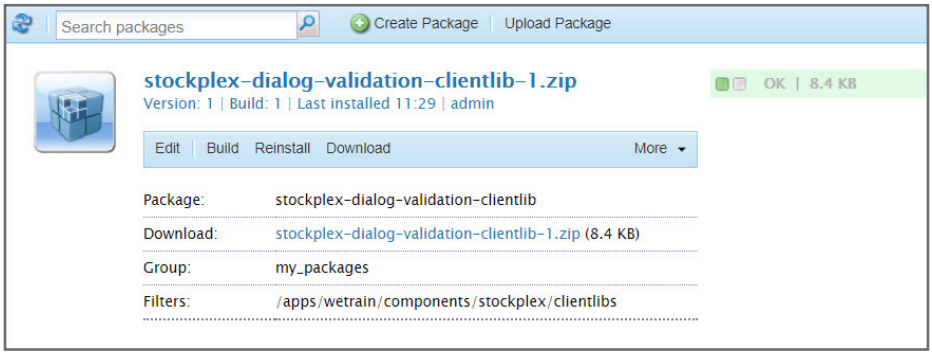
Task 4: Add Symbol Validation

Now that you have a basic dialog with two form fields, you need to create validation of the stock symbol. As the symbol description suggests, the user must enter a four-letter symbol. You can accomplish this by including client-side JavaScript with a client library. For convenience, you will upload the client library with the package manager.

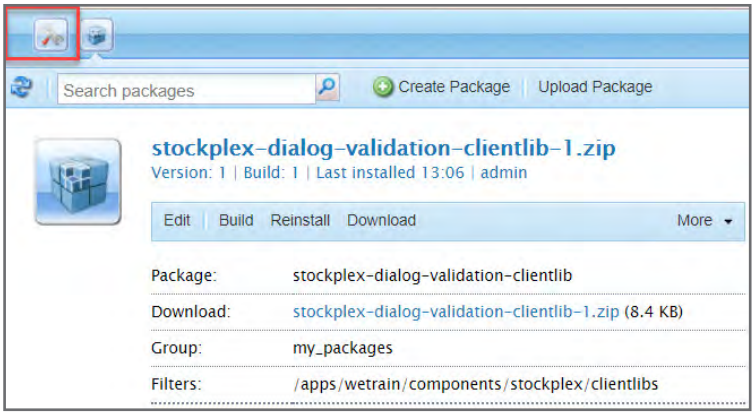
1. In your browser, open the AEM homepage and navigate to **Tools** > **Deployment** > **Packages** (http://localhost:4502/crx/packmgr/index.jsp) to open the package manager, as shown:

2. Click **Upload Package** in the actions bar. The **Upload Package** dialog opens.
3. Click **Browse** in the dialog. The **Open** dialog opens.

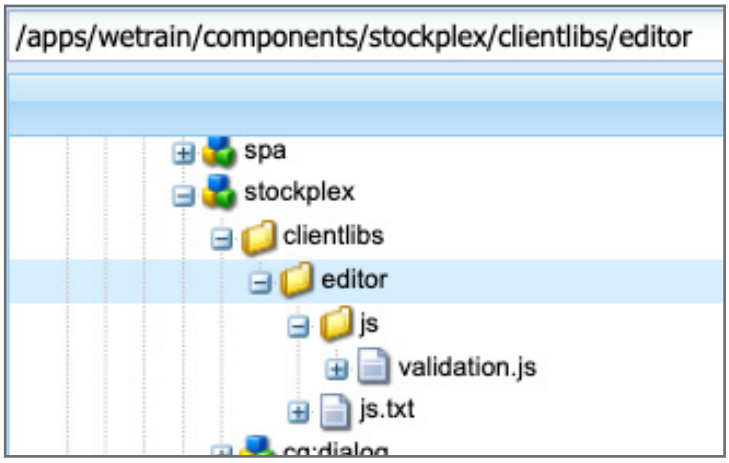
4. Navigate to **Exercise_Files-DWC/training-files/custom-cmps** folder, select the **stockplex-dialog-validation-clientlib-1.zip** package, and click **Open**.
5. Click **OK** in the Upload Package dialog to close the dialog.
6. Click **Install** in the package actions bar and then click **Install** again in the Install Package confirmation dialog to install the package just uploaded.



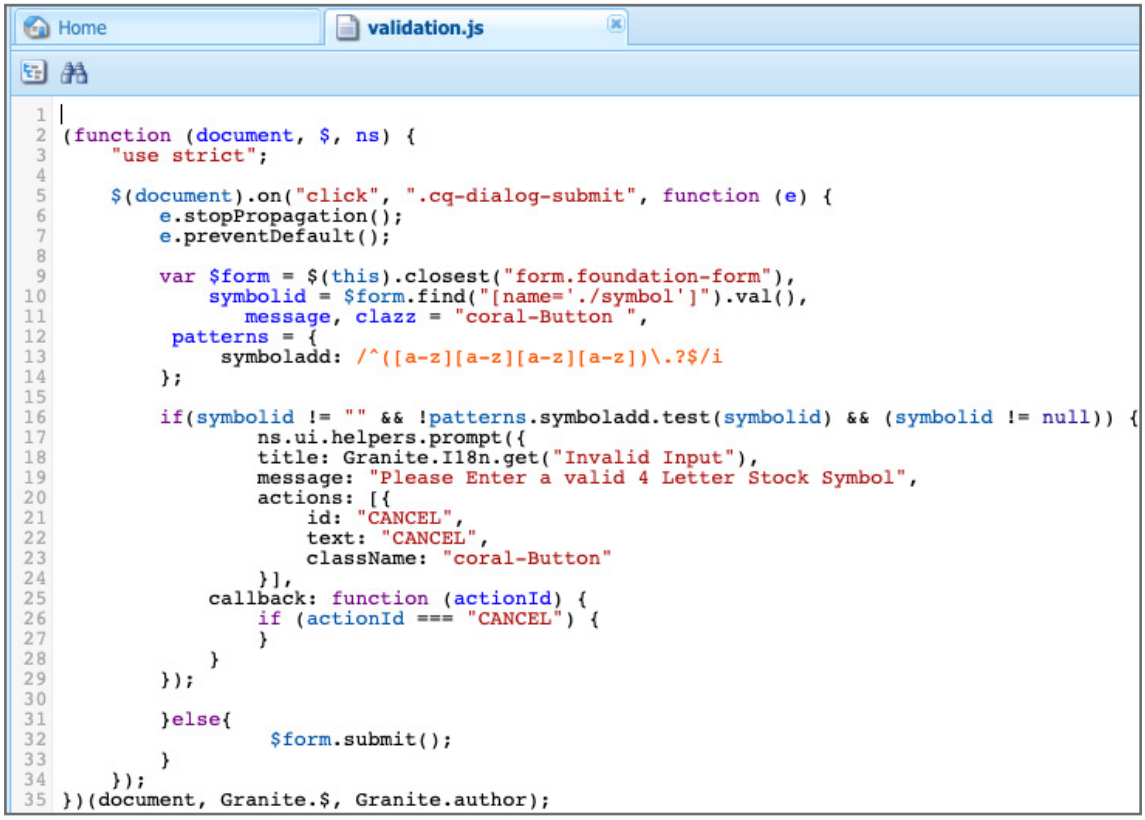
7. Click on the Develop icon (dragonfly icon) from the header bar, as shown. The CRXDE Lite page opens.



8. Navigate to **/apps/wetrain/components/stockplex**.
9. Expand the **clientlibs** folder. Notice that the content package installed a client library for the stockplex component, Editor: A client library containing a validation script for a dialog

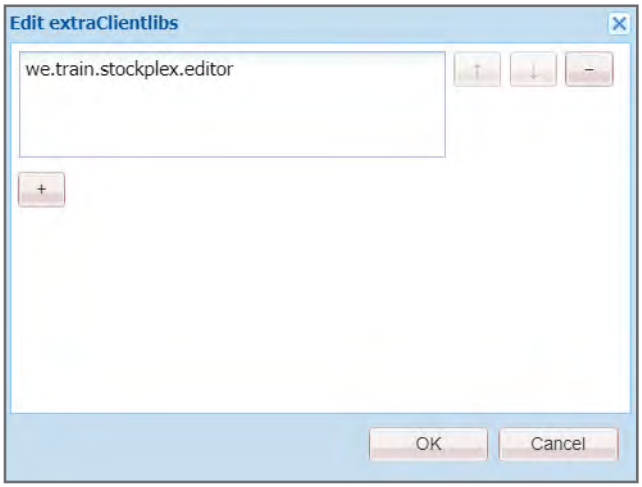


10. Open **validation.js** and examine the script.



To use the stockplex editor client library, you need to it to load when the dialog is opened. Do this by adding the category name to the dialog node.

11. In **CRXDE Lite**, navigate to **/apps/wetrain/components/stockplex**.
12. Select the **cq:dialog** node.
13. Double-click the **extraClientlibs property**. The **Edit extraClientlibs** dialog box opens, as shown:

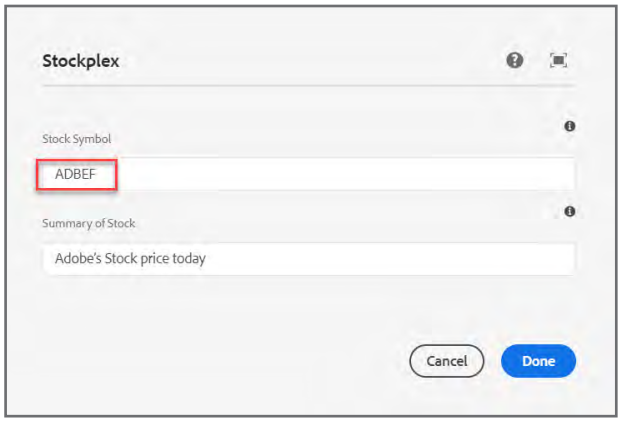


14. Verify that the value is **we.train.stockplex.editor** and click **OK**.
15. Save the changes.

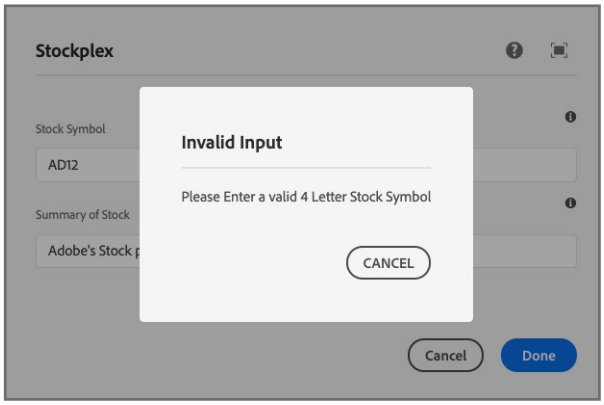
Task 5: Verify the Dialog Validation

1. In your browser, open the AEM homepage and navigate to **Sites** > **We.Train** > **Language Masters.**
2. Open the **en** page in the page editor.
3. If the **stockplex** component is not already on the page, drag one onto the page.
4. Click the **stockplex component** to open the Component toolbar.
5. Click the **wrench** to open the stockplex configure dialog.

6. In the Stock Symbol box, type more than four characters, for example, **ADBEF**, and click **Done.**



The Invalid Input dialog box opens with a message, as shown. This indicates that the validation script is working.



7. Try less than four characters. Try four numbers or a combination of numbers and letters. The validation script will only accept four alpha characters.

Task 6: Import the Changes to Your AEM Project

The last few tasks in this exercise were done directly in the repository, so you must export the changes from the repository and import them into your AEM project.

1. In your IDE, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components**.
2. Right-click on the **stockplex component folder** and click **Import from AEM Server.**

Exercise 3: Create a Content Policy to Control Component Behavior

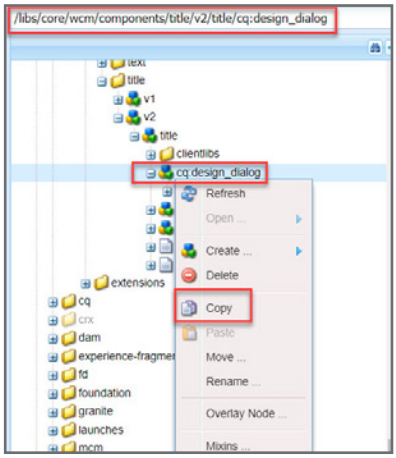
The stockplex component's functional requirements specify that template configurations need to be defined for optional detailed stock information. This can be accomplished by creating a design_dialog. As a result, the template author will be able to create a content policy for the stockplex component. In this exercise you will create a design dialog, import changes into the AEM project, update the script, then import policy changes into the AEM project.

Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service
- Existing stockplex component with a dialog

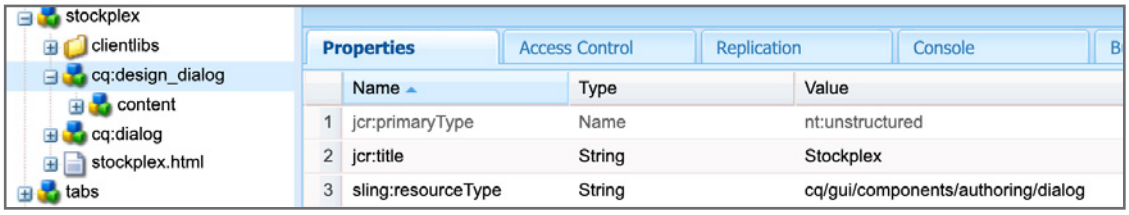
Task 1: Create a Design Dialog

1. In **CRXDE Lite**, navigate to **[/libs|/apps]/core/wcm/components/title/v2/title**.
2. Right-click on the **cq:design_dialog** node and click **Copy**, as shown:



3. Navigate to **/apps/wetrain/components**.

4. Right-click on the **stockplex** component node and click **Paste** to paste the copied **cq:design_dialog** node.



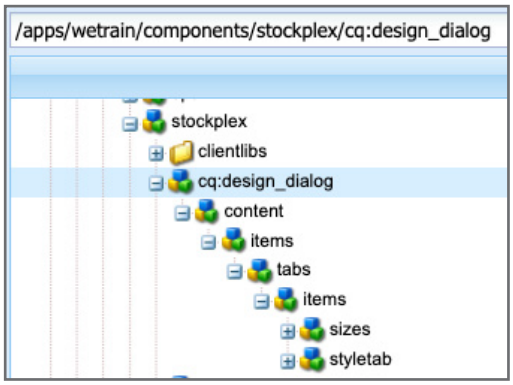
5. Click **Save All**.

6. Modify the properties of the **cq:design_dialog** node as shown:
   - **jcr:title** = **Stockplex**

7. **Delete** the following two properties:
   - extraClientlibs
   - helpPath

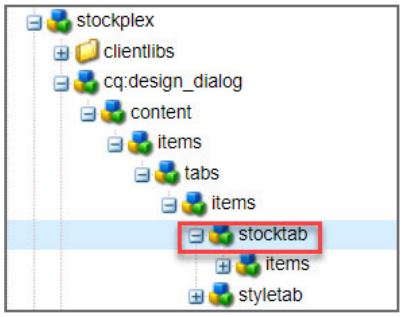
8. Click **Save All**.

9. Expand the **cq:design_dialog** node, as shown.

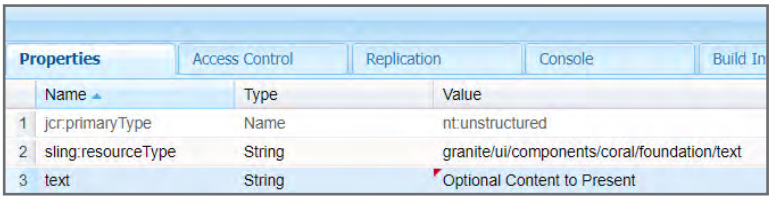


10. Navigate to **cq:design_dialog/content/items/tabs/items**.

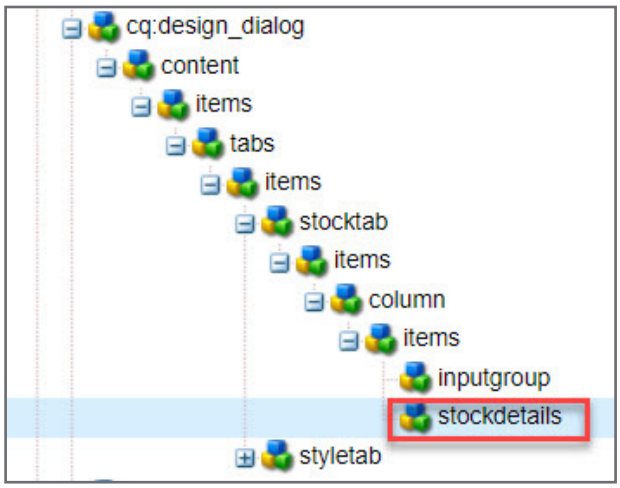
11. Select the **sizes** node and **rename** to **stocktab**, as shown:



12. Modify the **jcr:title** property on the **stocktab** node as shown:
    - **jcr:title** = **Stock Info**
13. Save the changes.
14. Continue to edit the **Stock Info** tab.
15. Navigate to **stocktab/items/column/items/inputgroup**.
16. Modify the **text** property on the inputgroup node as shown:
    - text = **Optional Content to Present**



17. Save the changes.
18. Under the **stocktab/items/column/items** node, **Delete** the following nodes:
    - field
    - type
19. Save the changes.
20. Under the **stocktab/items/column/items** node, select the **disableLink** node and **rename** it to **stockdetails:**



21. Modify the properties of the **stockdetails** node, as shown:

Name	Type	Value
fieldDescription	String	Include requestDate,upDown, openPrice,range high/low, and volume
name	String	./showStockInfo
text	String	Include Stock Information

Note: The name property is case sensitive! Make sure the value is exactly the same.

22. Save the changes.

Task 2: Import Changes into the AEM Project

The work in the previous task has been done directly in the repository, so you must export the changes from the repository and import them into your AEM project.

1. In your IDE, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components**.
2. Right-click on the **stockplex** component folder and select **Import from AEM Server.**

Task 3: Update the Script

1. In the Exercise_Files-DWC folder, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/stockplex**.
2. Copy contents of the **policy_stockplex.html** file and paste it into the **stockplex.html** file in your IDE.
3. Save the changes.
4. Examine the script to see how the **Stockplex** component renders the properties based on design element values:

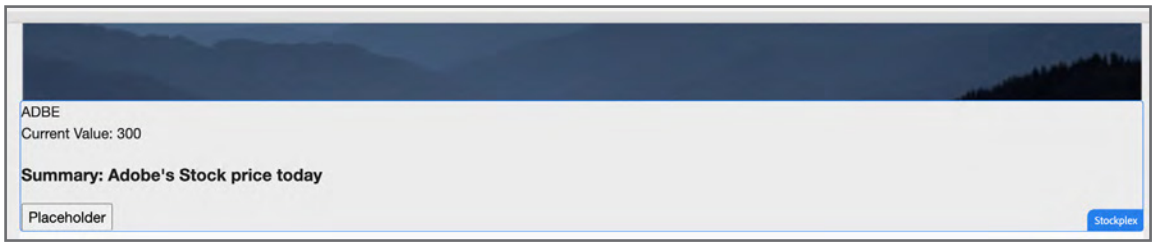
ui.apps > src > main > content > jcr_root > apps > wetrain > components > stockplex > stockplex.html > sly

```
<!--/* Similar to dialogs, policy values from design dialogs can
    be retrieved with a global object called 'currentStyle'.
    */-->
    <div data-sly-use.template="core/wcm/components/commons/v1/templates.html"
    data-sly-test.hasContent="${properties.symbol}"
    class="">

    <div class="">${properties.symbol}</div>
    <div class="">Current Value: 300</div>

    <div data-sly-test.summary="${properties.summary}">
        <h3>Summary: ${properties.summary}</h3>
    </div>

    <div class="">
        <a href="#">
            <button>Placeholder</button>
        </a>
    </div>

    <!--/* The global object currentStyle can be used to retrieve configurations
        set at the policy level with a cq:design_dialog */-->
    <div class="" data-sly-test.summary="${currentStyle.showStockInfo}">
        <ul data-sly-list.sInfo="${['Request Date','Open Price', 'Range High', 'Range Low', 'Clo
            <li class="">
                <span class="">${sInfo}: value</span>
            </li>
        </ul>
    </div>
</div>

<!-- If there is no stock symbol added to the dialog, create a component placeholder -->
<sly data-sly-call="${template.placeholder @ isEmpty=!hasContent, classAppend='cmp-stockplex'}">
```

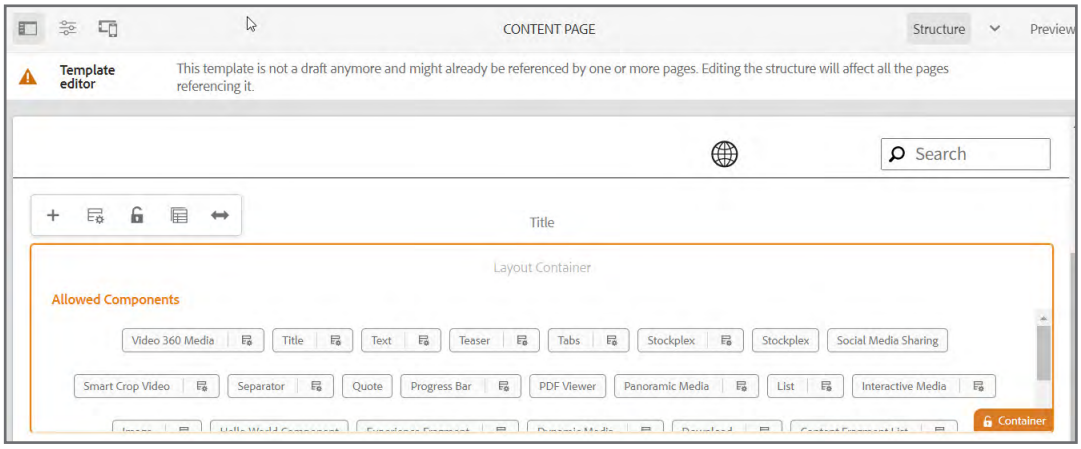
Task 4: Create a Content Policy

1. To verify the project deployment, in your browser, open the AEM homepage and navigate to **Sites > We.Train > Language Masters**.

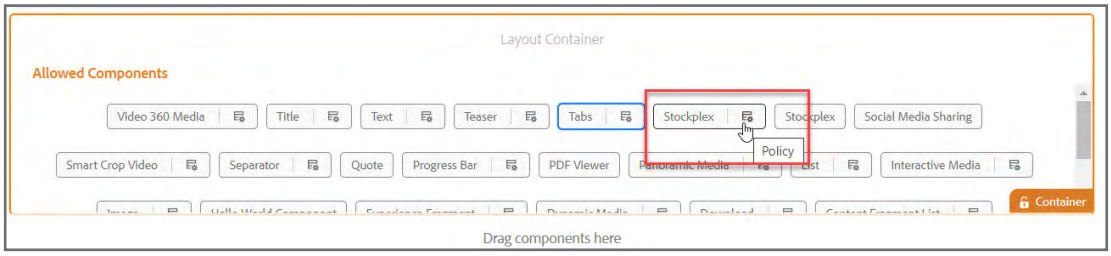
2. Open the **en** page in the page editor.
   You should see that the Stockplex component on the page has changed, as compared to the previous exercise. The Stock Info is no longer shown.



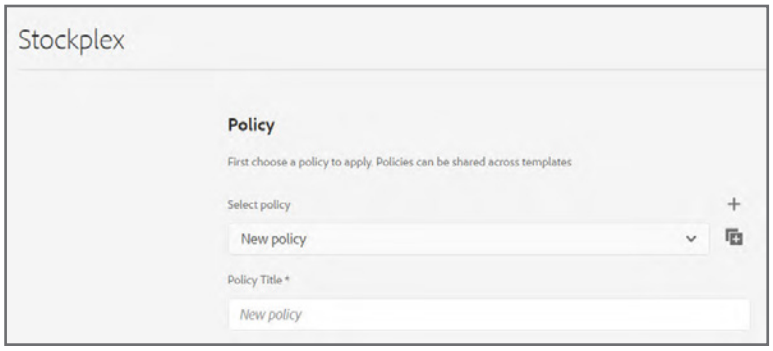
3. Click the **Page Information** menu and then click **Edit Template**. The template editor opens the **CONTENT PAGE** template.



4. Click the **policy** icon in the **Stockplex** component to open the Policy dialog.

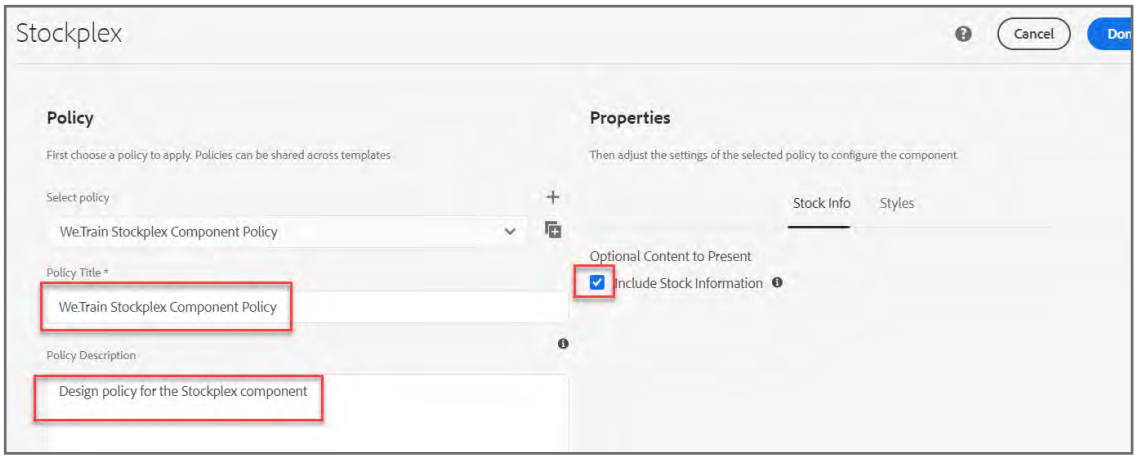


5. Verify that the **Select policy** field contains **New Policy**, as shown:



6. Enter the following information into the policy dialog, as shown:

Field Name	Value
Policy Title	We.Train Stockplex Component Policy
Policy Description	Design policy for the Stockplex component
Include Stock Information	Checked



7. Once a new policy is created, click **Done**.

8. Return to the browser tab with the **en** page and refresh. The **stockplex** component now displays the Stock Info.

Task 5: Import Policy Changes into AEM Project

The work in the previous task was done directly in the repository, so you must export the changes from the repository and import them into your AEM project.

1. In your IDE, navigate to **ui.content/src/main/content/jcr_root/conf/wetrain/settings**
2. Right-click on the **wcm** folder and click **Import from AEM Server.**

Exercise 4: Add a Default Design and Optional Styles

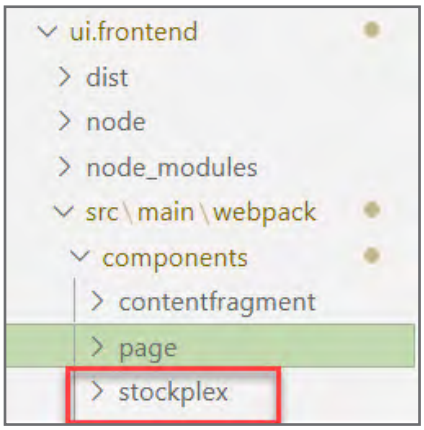
The functional requirements specify that the component should follow the We.Train design and an author should have the ability to change the style of the text within the component. In this exercise, you will create stylesheets for each optional style and then add those styles to the content policy of the stockplex component.
In this exercise you will add component styles to ui.frontend, update the component script to use the styles, then deploy and verify.

Prerequisites:

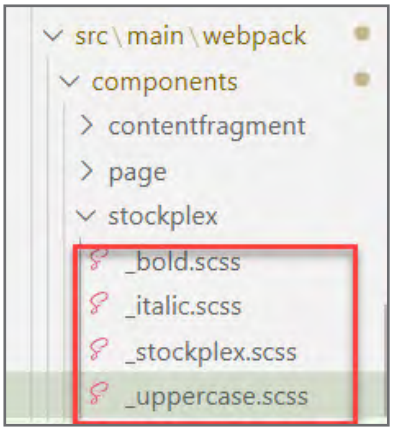
- Local author service running
- A Maven project imported into your IDE and installed on the author service
- Existing stockplex component with dialog and design dialog

Task 1: Add Component Styles to ui.frontend

1. In your IDE, navigate to **ui.frontend/src/main/webpack/components**.
2. Create a folder named **stockplex** under the components folder, as shown:

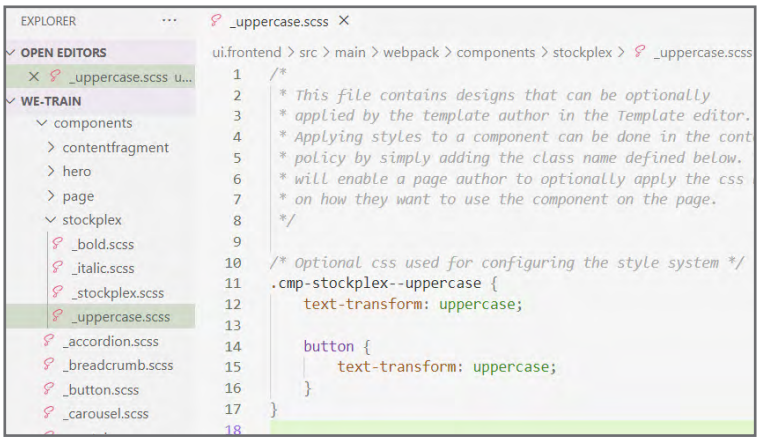


3. Create four files under the **stockplex** folder, with the names shown here:
   - **_bold.scss**
   - **_italic.scss**
   - **_stockplex.scss**
   - **_uppercase.scss**



Note: _stockplex.scss contains the default design. The rest of the files are optional styles a template author can configure in the styles tab of a content policy.

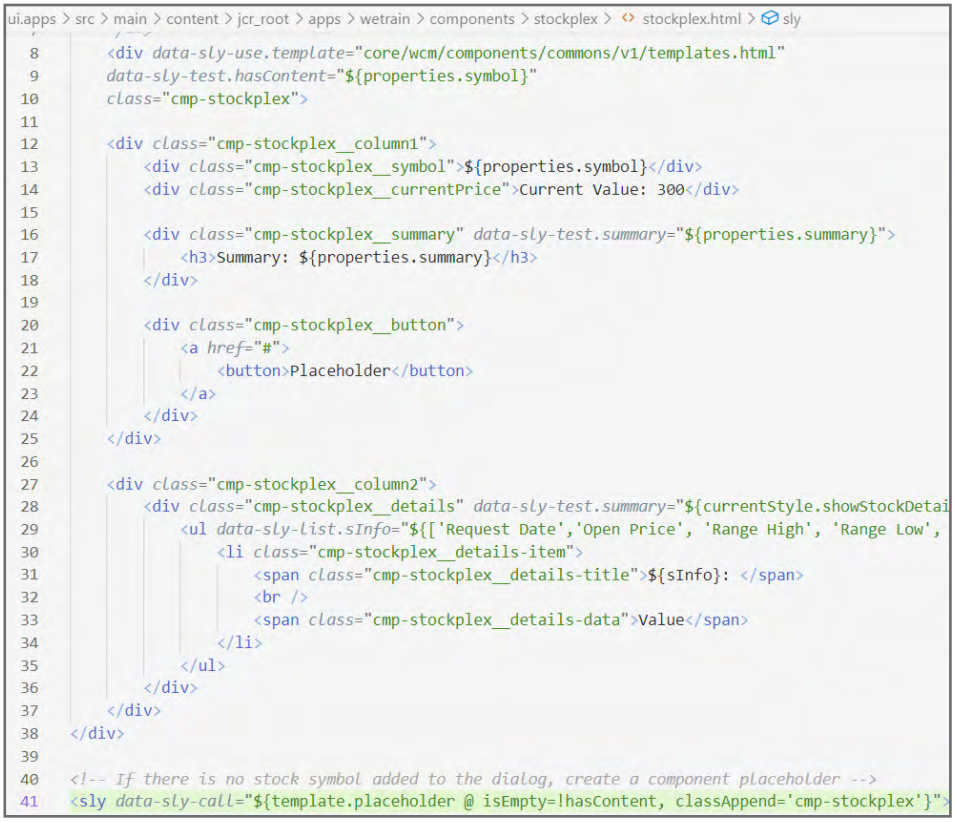
4. In your Exercise_Files-DWC, navigate to **ui.frontend/src/main/webpack/components/stockplex.**
5. Open each file under the **stockplex** folder and paste the contents in the corresponding file that you created in your IDE.



6. Examine the four files to familiarize yourself with the style classes.

Task 2: Update the Component Script to Use the Styles

1. In the Exercise_Files-DWC folder, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/stockplex**.
2. **Copy** contents of the **styles_stockplex.html** file and **paste** it into the **stockplex.html** file, as shown:



3. Save the changes.
4. Examine the script to see how the style classes, added in the previous task, are used.

Task 3: Deploy and Verify

The IDE should have synched any saved script changes to the project. However, we execute this build to deploy the changes made to the webpack app in ui.frontend.

1. In your IDE, open a terminal window.

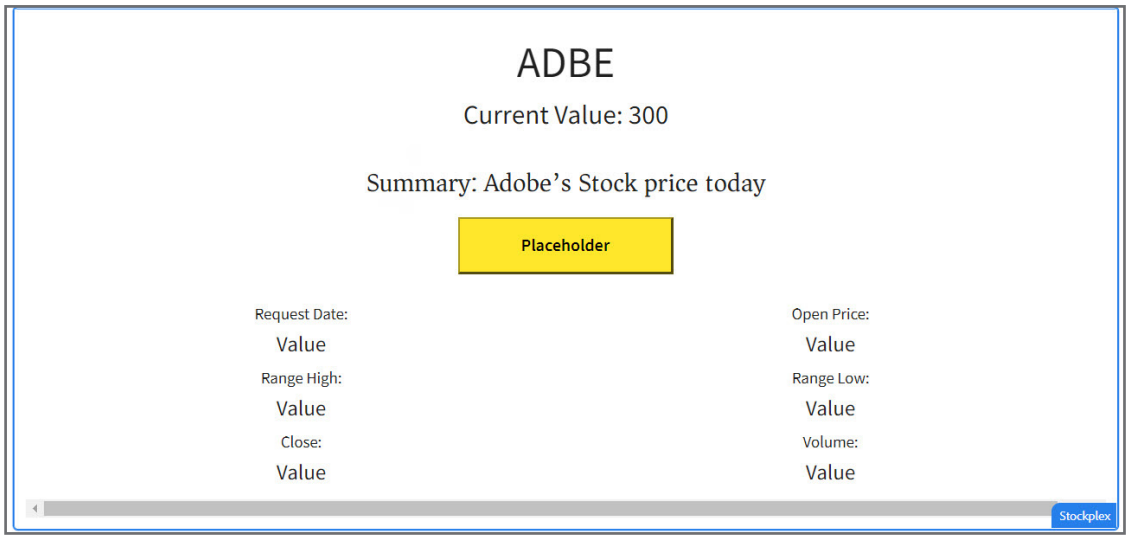
Note: As an alternative to the IDE terminal window you could open an OS terminal window in your <AEM Project> folder.

2. Type in the following command to deploy your project to AEM:

```
mvn clean install -PautoInstallSinglePackage
```

3. To verify the project deployment, in your browser, open the AEM homepage and navigate to **Sites** > **We.Train** > **Language Masters**.

4. Open the **en** page in the page editor. Verify the style classes are rendered by the script, as shown:



Exercise 5: Add Styles to a Content Policy

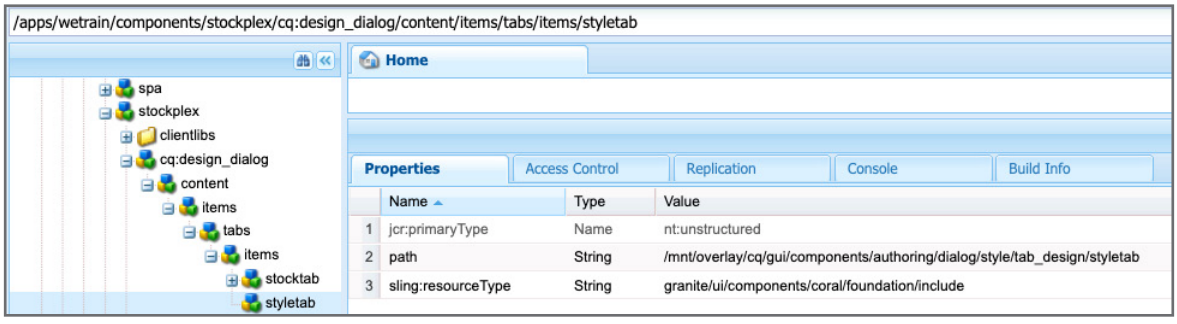
To give authors the ability to optionally select different styles of the component, the design_dialog needs to have a styletab configured with the optional styles the authors can choose.
In this exercise, you will investigate the styletab functionality in the design_dialog, update the content policy to include styles, then Import policy changes into AEM project.

Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service
- Existing stockplex component with dialog and design dialog
- Existing stockplex content policy
- Stockplex folder and style files added to ui.frontend module

Task 1: Investigate the styletab Functionality in the design_dialog

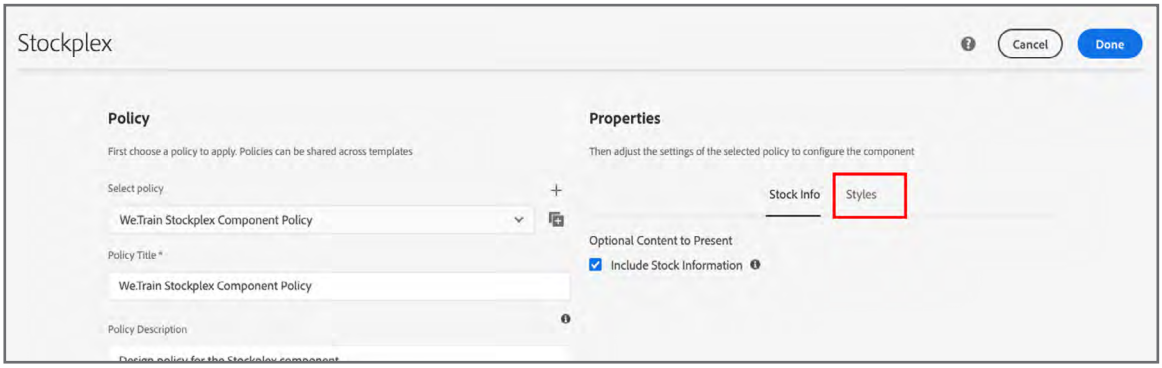
1. In CRXDE Lite, navigate to **/apps/wetrain/components/stockplex/cq:design_dialog/content/items/tabs/items**.
2. Select the **styletab** node. This node enables the Style System functionality for the stockplex component.



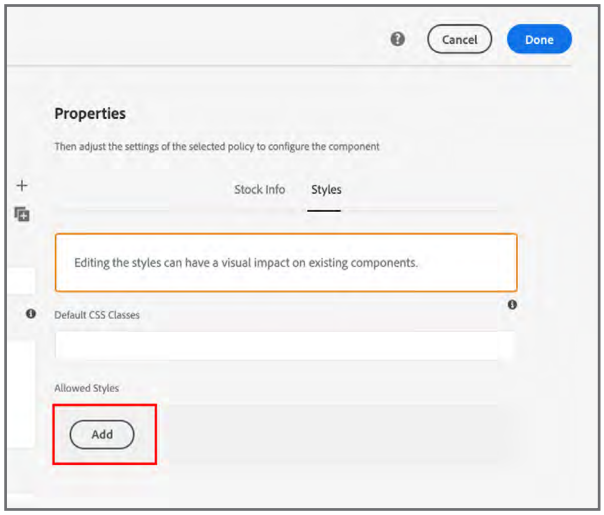
Note: This tab was copied over from the title component in a previous exercise. Every core component has this same tab to enable the style system.

Task 2: Update the Content Policy to Include Styles

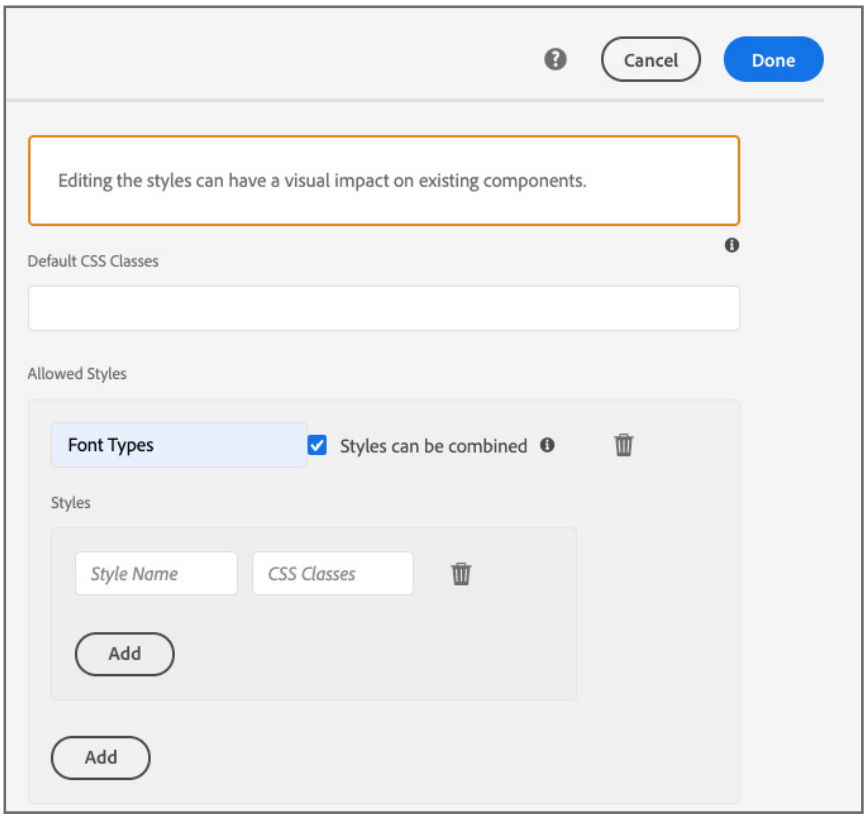
1. In your browser, open the AEM homepage and navigate to **Sites** > **We.Train** > **Language Masters**.
2. Open the **en** page in the page editor.
3. Click the **Page Information** menu and then click **Edit Template**. The Content Page template opens in the template editor.
4. Click the policy icon in the stockplex component to open the policy editor.



5. Click the **Styles** tab in the **Properties** pane.



6. Click the **Add** button below the **Allowed Styles** field.
7. In the **Group Name** box, type **Font Types**, and select the **Styles can be combined** checkbox.
8. Click the **Add** button that is below the **Styles** field, as shown. A new style field is created.

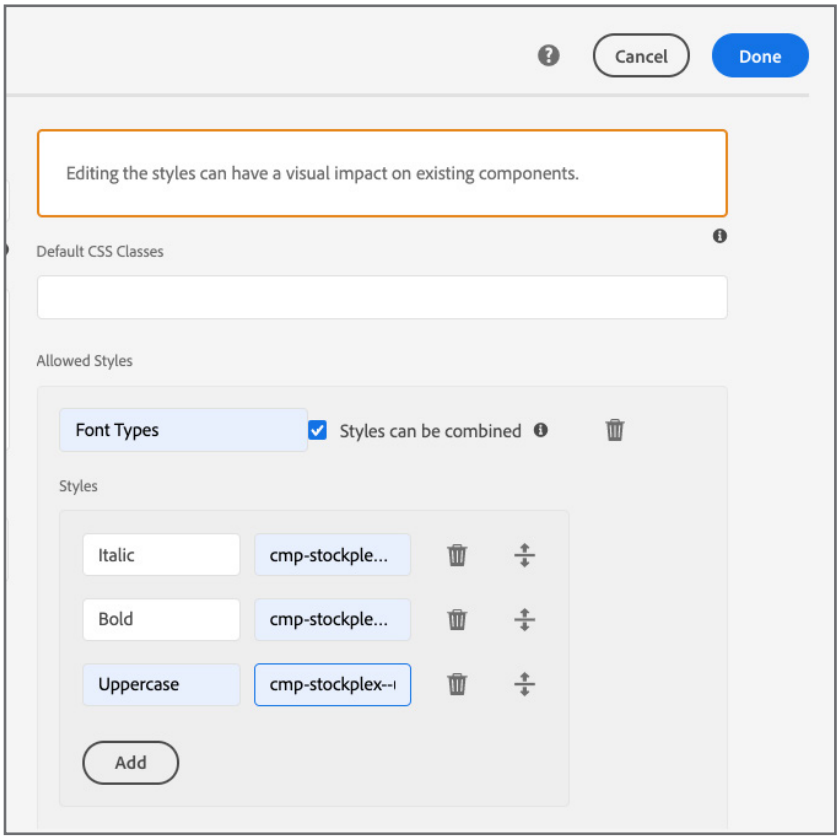


9. Enter the following three styles:

Style Name	CSS Classes
Italic	cmp-stockplex--italic
Bold	cmp-stockplex--bold
uppercase	cmp-stockplex--uppercase

Note: Make sure you have double hyphen added at the end of these three CSS Classes. For example **cmp-stockplex--bold**.

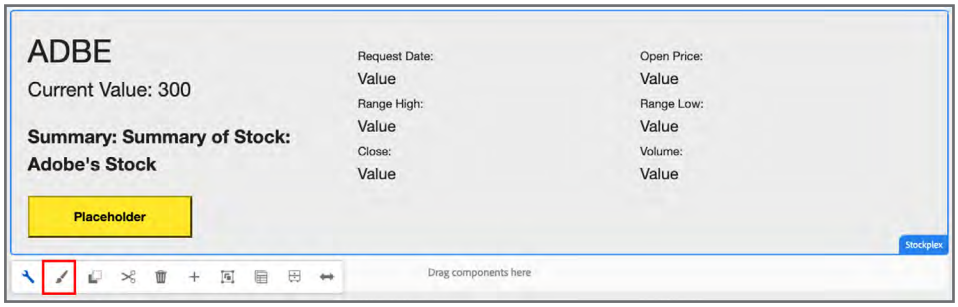
Your styles should look similar to the one shown in the following screenshot:



10. Click **Done.** The policy editor closes.

Now that you have the CSS developed by the designer, and the style system is added by the template editor, authors can use the style system.

11. In a browser, open the **en** page tab.
12. Click the **stockplex component** to open the Component toolbar. A brush (the Style System icon) has appeared in the toolbar, as shown:



13. Click the style icon to expand the styles menu.
14. Select styles from the dropdown menu and observe the changes to the component output, as shown:

Task 3: Import Policy Changes into AEM Project

The work in the previous task was done directly in the repository, so you must export the changes from the repository and import them into your AEM project. Import the updated policy.

1. In your IDE, navigate to **ui.content/src/main/content/jcr_root/conf/wetrain/settings.**
2. Right-click on the **wcm** folder and select **Import from AEM Server.**

Exercise 6: Create Translation Dictionaries

Some of the We.Train components render string constants. To meet the functional requirements, those strings must be translated into the supported languages. In this exercise you will define a translation string library for the string constants.
In this exercise you will update the component script, update the translation file with the translated strings, then deploy and verify.

Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service
- Existing stockplex component with dialog and design dialog
- Content policy defined to show Stock Details

Task 1: Update the Component Script

1. In the Exercise_Files-DWC folder, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/stockplex**.

2. Copy contents of the **i18n_stockplex.html** file and **paste** it into the **stockplex.html** file that you just created.

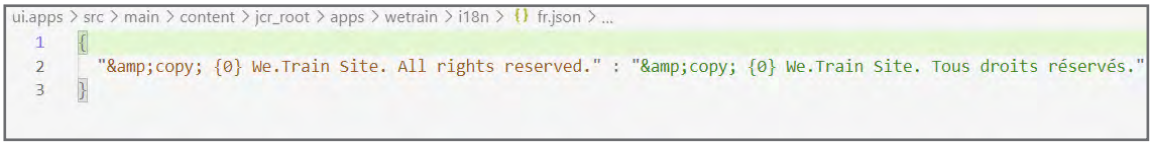
3. Save the changes.

4. Examine the script to see how string constants that appear in scripts can be translated.

Task 2: Update the Translation File with the Translated Strings

1. In your IDE, navigate to **/apps/wetrain/i18n**. This is the folder that holds all the translation information for the **wetrain** project.

2. The AEM Project Archetype generated an initial French translation file: **fr.json**. Open the **fr.json** file.



3. Take notice of the structure of the key:value pair separated by a colon (**:**), where the
   - **key** = the string to be translated
   - **value** = the translated string

4. In the Exercise_Files-DWC folder, navigate to **ui.apps/src/main/content/jcr_root/apps/i18n**.
5. Copy contents of the **fr.json** file and **paste** it into the existing **fr.json** file.

ui.apps > src > main > content > jcr_root > apps > wetrain > i18n > {} fr.json >

```
{
  "Summary:": "Sommaire:",
  "Current Value:": "Valeur courante:",
  "Headless JSON View": "Vue JSON sans tête",
  "52 Week Low": "Min. année écoulée",
  "52 Week High": "Max. année écoulée",
  "Sector": "Secteur",
  "Company": "Société",
  "Volume": "Volume",
  "Range Low": "Fourchette basse",
  "Range High": "Fourchette haute",
  "Open Price": "Prix d'ouverture",
  "Up/Down": "Haut/Bas",
  "Request Time": "Heure de demande",
  "Request Date": "Date de demande"
}
```

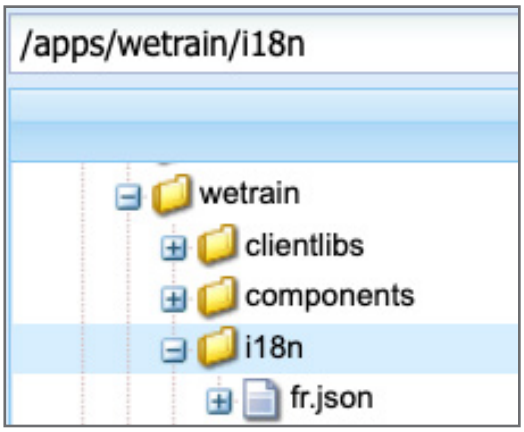
6. Save the changes.

Task 3: Deploy and Verify

1. In your IDE, open a terminal window.
2. Type in the following command to deploy your project to AEM:

```
mvn clean install -PautoInstallSinglePackage
```

3. In CRXDE Lite, navigate to **/apps/wetrain/i18n**. This is the folder that contains all of the translation information for the **wetrain** project.



4. Select the **fr.json** file and take note of the following properties:

Property Name	Value	Description
jcr:language	fr	Sets the language to French. Notice the use of the ISO code
jcr:mixinTypes	mix:language	Identifies this file as a language file

5. Open the **fr.json** file to verify that the contents were deployed.
6. In a browser, navigate to **Sites** > **We.Train** > **Language Masters.**
7. Using the skills you learned previously, create a new page with the following attributes:

Attribute	Value
Template	Content Page
Title	French
Name	fr

8. Open the **French** page in the page editor.
9. Drag a **Stockplex component** on to the page.
10. Open the component dialog and enter the following values and click **Done**:
    - Stock Symbol: **ADBE**
    - Summary of Stock: **Adobe's Stock**

(Optional) Exercise 7: Create Additional Supported Languages

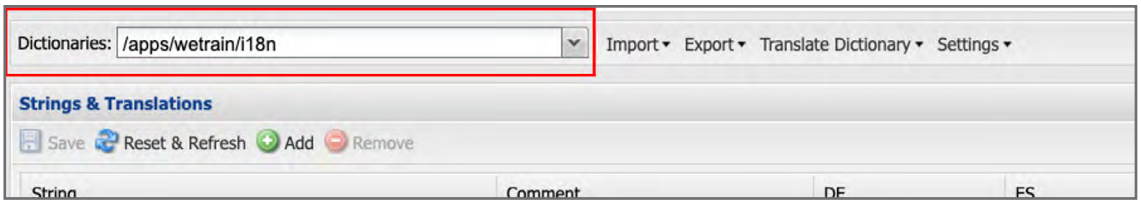
Now that stockplex.html supports French translations, this can be easily expanded to any number of languages. In this exercise, you will create Italian and Spanish translations to support Stockplex.
In this exercise, you will use the ldiff dictionary to add strings, add additional language strings, then import changes into your AEM project.

Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service
- Existing stockplex component with dialog and design dialog
- I18n folder configured with fr.json and French string translations

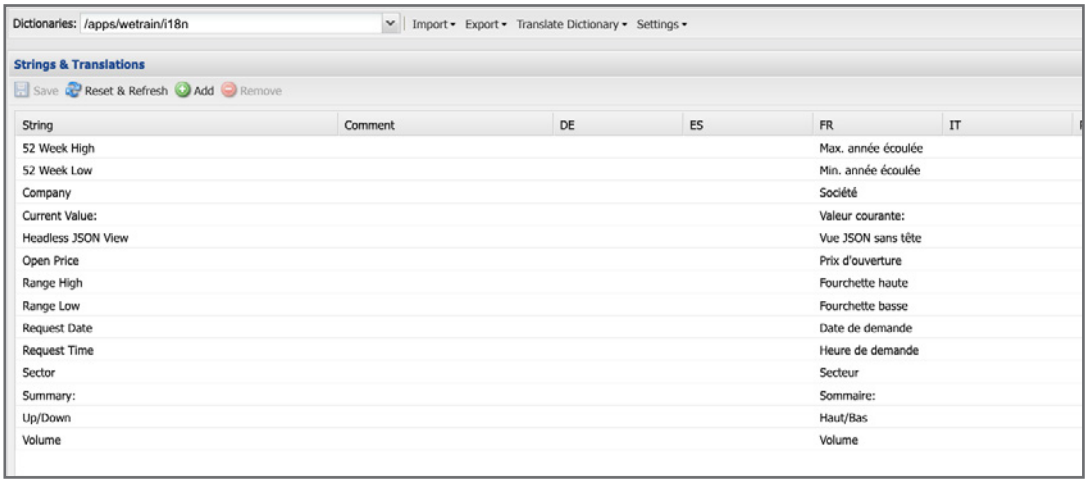
Task 1: Use the ldiff Dictionary to Add Strings

1. In a new browser tab, enter http://localhost:4502/libs/cq/i18n/translator.html to open the **Strings and Translation** page, as shown:

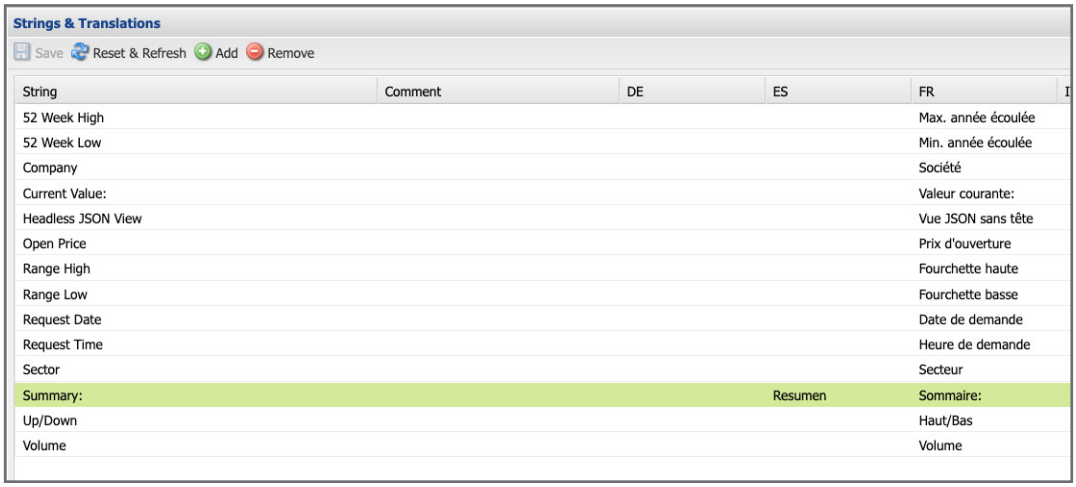


2. In the Dictionaries field, select **/apps/training/i18n** from the dropdown menu. Notice the

translated strings in the **FR** column. These strings came from the **fr.json** file.

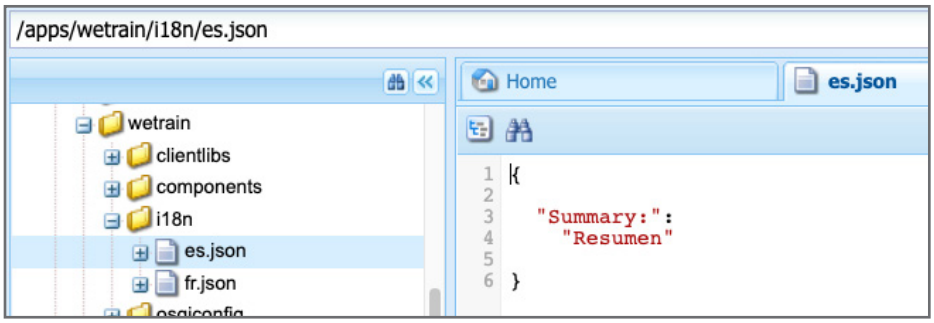


3. In the line that starts with **Summary:**, click in the ES column and enter the Spanish translation for Summary, which is **Resumen:**, and **Save**.



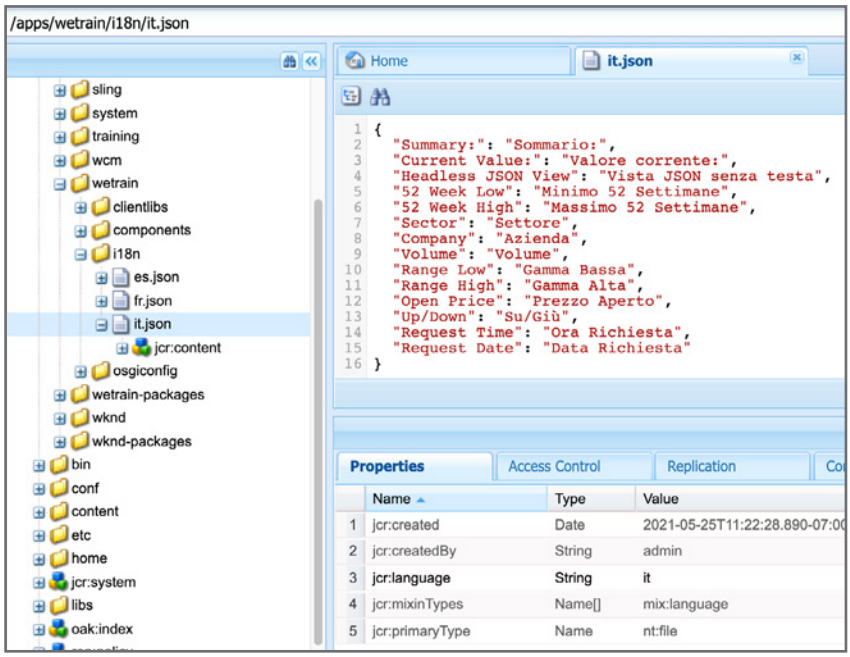
4. In another browser tab, open **CRXDE Lite**. Navigate to **/apps/wetrain/i18n.**

5. Notice that an **es.json** file has appeared. It contains the string that you entered into the dictionary. The es.json file has similar properties to the fr.json. The difference is that the **jcr:language** property is set to es, instead of fr.



Task 2: Add Additional Language Strings

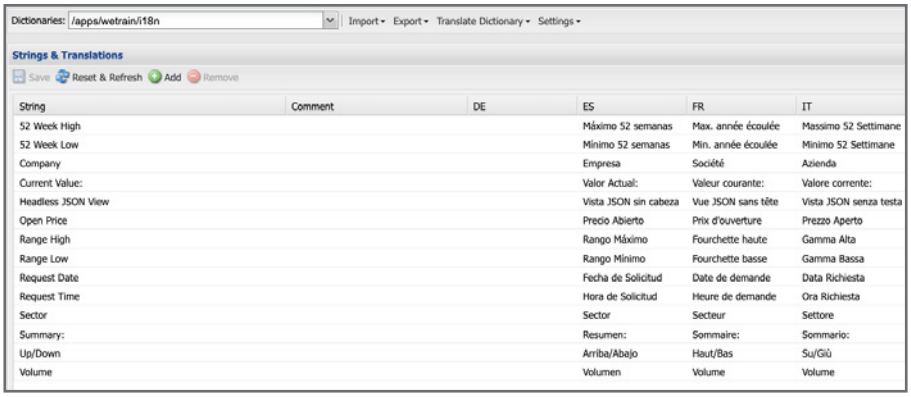
1. In **CRXDE Lite**, navigate to **/apps/i18n**.
2. Right-click on the **es.json** file and click **Copy.**
3. Right-click on the **i18n** node and click **Paste**.
4. Rename the pasted file to **it.json.**
5. Save the changes.
6. Modify the value of the **jcr:language property** of the it.json file to **it.**
7. In the Exercise_Files-DWC folder, navigate to **ui.apps/src/main/content/jcr_root/apps/i18n**.
8. Replace the contents of the Italian and Spanish .json files using the files in the Exercise_Files-DWC.



9. Save the changes.

10. Return to the Translation dictionary tab, or enter http://localhost:4502/libs/cq/i18n/translator.html in a new tab, to open the **Strings and Translation** page.

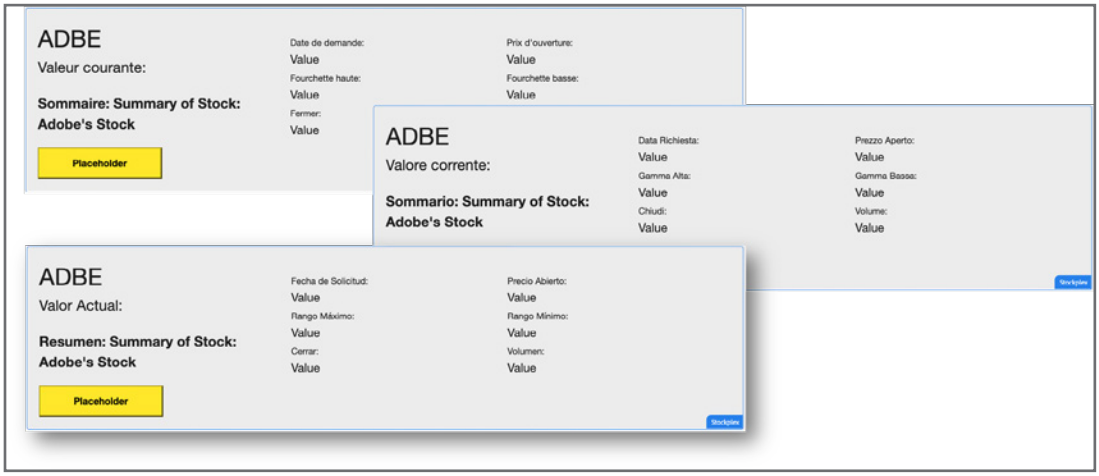
11. In the Dictionaries field, select **/apps/training/i18n** from the dropdown menu. Notice the translated strings in the ES, FR and IT columns.



String	Comment	DE	ES	FR	IT
52 Week High			Máximo 52 semanas	Max. année écoulée	Massimo 52 Settimane
52 Week Low			Mínimo 52 semanas	Min. année écoulée	Minimo 52 Settimane
Company			Empresa	Société	Azienda
Current Value:			Valor Actual:	Valeur courante:	Valore corrente:
Headless JSON View			Vista JSON sin cabeza	Vue JSON sans tête	Vista JSON senza testa
Open Price			Precio Abierto	Prix d'ouverture	Prezzo Aperto
Range High			Rango Máximo	Fourchette haute	Gamma Alta
Range Low			Rango Mínimo	Fourchette basse	Gamma Bassa
Request Date			Fecha de Solicitud	Date de demande	Data Richiesta
Request Time			Hora de Solicitud	Heure de demande	Ora Richiesta
Sector			Sector	Secteur	Settore
Summary:			Resumen:	Sommaire:	Sommario:
Up/Down			Arriba/Abajo	Haut/Bas	Su/Giù
Volume			Volumen	Volume	Volume

12. Using the skills that you learned in the last exercise, create Spanish and an Italian pages.

13. Add a Stockplex component to each page and observe the translations.



Task 3: Import Changes into Your AEM Project

The work in the previous task has been done directly in the repository, so you must export the changes from the repository and import them into your AEM project. Import the updated policy.

1. In your IDE, navigate to **ui.apps/src/main/content/jcr_root/apps**.
2. Right-click on the **i18n folder** and click **Import from AEM Server.**

Exercise 8: Add a Sling Model as Business Logic

The stockplex component functional requirements specify that the component display detailed stock information gathered from an external source. There are many ways you could pull content in from a third party depending on how up-to-date the data needs to be as well as how performant the page needs to be. In this scenario, performance is the priority and near real-time data is acceptable. This exercise will use several Java backend techniques to import the stock data into the JCR and retrieve it for stockplex.

In this exercise you will install the sling model, install backend configurations and update the component script.

To learn more about the Java code used in this exercise, consider taking the Extend and Customize Adobe Experience Manager course.

Note: For training purposes, the stock data is gathered from an external git repository.

Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service
- Existing stockplex component with dialog and design dialog
- Content policy defined to show stock details

Task 1: Install the Sling Model

Note: Do this task only if you have not already copied the Exercise_Files-DWC/core/**src** folder into your < AEM Project>.

1. Using the file system, in the Exercise_Files-DWC, copy the **core/src** folder.
2. Using the file system, navigate to your <AEM Project> and replace the **core/src** folder with the copied folder.

Note: The core module contains all the Java code that reaches out to the third party source, saves the data into the JCR, and includes a Sling Model to retrieve the data.

Task 2: Install Backend Configurations

1. In your browser, open the AEM homepage and navigate to **Tools** > **Deployment** > **Packages** (http://localhost:4502/crx/packmgr/index.jsp) to open the package manager, as shown:

2. Click **Upload Package** in the actions bar. The **Upload Package** dialog opens.
3. Click **Browse** in the dialog. The **Open** dialog opens.
4. Navigate to **Exercise_Files-DWC/training-files/custom-cmps** folder and select the **stockplex-User-and-MappingConfig.zip** package and click **Open**.
5. Click **OK** in the Upload Package dialog to close the dialog.
6. Click **Install** in the package actions bar and then click **Install** again in the Install Package confirmation dialog to install the package just uploaded.

Task 3: Update the Component Script

1. In your IDE, navigate to **wetrain/src/main/content/jcr_root/apps/wetrain/components/stockplex**.
2. Open the **stockplex.html** file.
3. In the Exercise_Files-DWC folder, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/stockplex**.

4. Copy contents of the **slingmodel_stockplex.html** file and paste it into the **stockplex.html** file.

5. Examine the **stockplex.html** file:
   - Line 7: Returns an object named 'stockplex' from the model helper class.
   - Line 12-18: Properties have been updated to use the returned values from the model.
   - Line 20-25: Button to show JSON export of the page containing the component.

6. Save the changes.

7. Examine the script to determine the differences in how the properties values are captured when using the Sling Model as a helper class.

Task 3: Deploy and Verify

1. In your IDE, open a terminal window.

Note: You must deploy to AEM. Even though the IDE has synchronized the script, you must build the project to deploy the OSGI bundle containing the Sling Model.

2. Type in the following command to deploy your project to AEM:

```
mvn clean install -PautoInstallSinglePackage
```

3. Using a browser, navigate to **Sites** > **stocks**.

4. Click the Create button and select **Folder**.
5. Name the folder **ADBE**. This creates a configuration to write the data, which the Stockplex Sling Model will read, into the repository. The scheduled job that runs in the background may take up to 2 minutes to create the data.

6. Using a browser, navigate to **Sites** > **We.Train** > **Language Masters**.
7. Open the **en** page in the page editor.

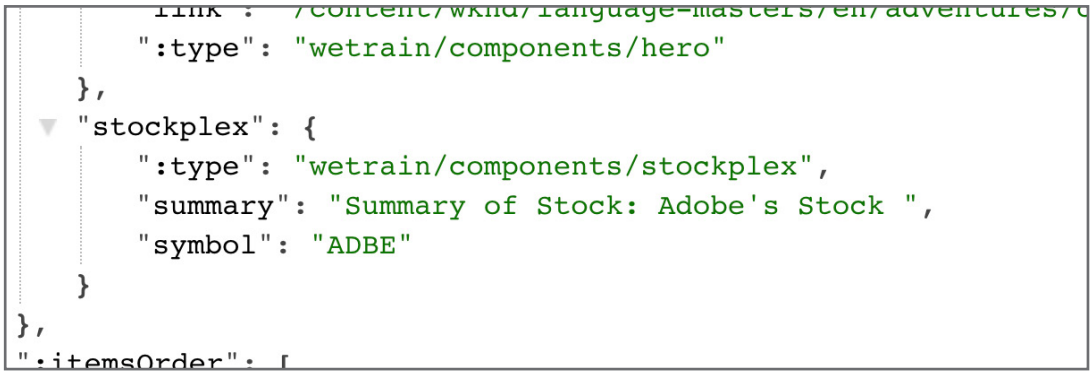
To view the page as JSON:

8. Use the Page Information menu to **View as Published.** The page, without the editor, opens in a new tab.
9. In the new tab, click the **Headless JSON View** button. The JSON rendering of the page opens. Notice the URL has changed. The .model selector has been added to the URL. Go back to the stockplex.html script and find the section that adds the selector to the URL.

```
20          <div class="cmp-stockplex__button">
21              <a href="${currentPage.path @ selectors='model', extension='json'}">
22                  <button>${'Headless JSON View' @ i18n}</button>
23              </a>
24          </div>
25      </div>
26      <div class="cmp-stockplex__column2">
27          <div class="cmp-stockplex__details" data-sly-test="${stockplex.showStockInfo}">
```

20-25 - Creates button to reload the page for JSON output by removing .html and adding .model.json

10. To find the rendering of the Stockplex component, search for **"stockplex": {** on the page.



Adobe Client Datalayer

The open-source Adobe Client Data Layer (ACDL) is an Event-Driven Data Layer (EDDL) that reduces the effort to instrument websites by providing a standardized method to expose and access any kind of data for any script. It consists of a JavaScript client-side event-driven data store that can be used on web pages to:

- Collect data about what the visitors experience on the web page
- Communicate this data to digital analytics and reporting servers

Core Component Events

The ACDL is fully integrated into the Core Components for easy use with AEM. There are a number of events that Core Components trigger via the Data Layer. The best practice for interacting with the Data Layer is to register an event listener and then take an action based on the event type and/or component that triggered the event. This will avoid potential race conditions with asynchronous scripts.

- **cmp:click** - Clicking a clickable element (an element that has a data-cmp-clickable attribute) causes the data layer to trigger a cmp:click event.
- **cmp:show** and **cmp:hide** - Manipulating the accordion (expand/collapse), the carousel (next/previous buttons), and the tabs (tab select) components causes the data layer to trigger cmp:show and a cmp:hide events respectively. A cmp:show event is also dispatched on page load and is expected to be the first event.
- **cmp:loaded** - As soon as the Data Layer is populated with the Core Components on the page, the Data Layer triggers a cmp:loaded event.

Collect Data for Adobe Analytics and Adobe Target

Paired with Adobe Experience Manager (AEM), Adobe Analytics and Adobe Target, the ACDL becomes the foundation of a very powerful and flexible tool set for gaining insights into your digital experiences. As mentioned earlier, the Adobe Client Data Layer is an event driven data layer. For example, when an AEM page is loaded in to the browser, the page-rendering component's data layer is also loaded. The data layer load will trigger an event: cmp:show. Using Adobe Launch, you can:

- Define an event-driven rule that will triggered based on the cmp:show event
- Map the page data layer properties to Data Elements
- Trigger a Page View beacon in Adobe Analytics

Using the Data Layer

The following methods are available to act on the data layer object:

- push(): Adds items to the data layer
- getState(): Returns the merged state of all pushed data.
- addEventListener: Sets up a function that will be called whenever the specified event is triggered.
- removeEventListener: Removes an event listener previously registered with addEventListener().

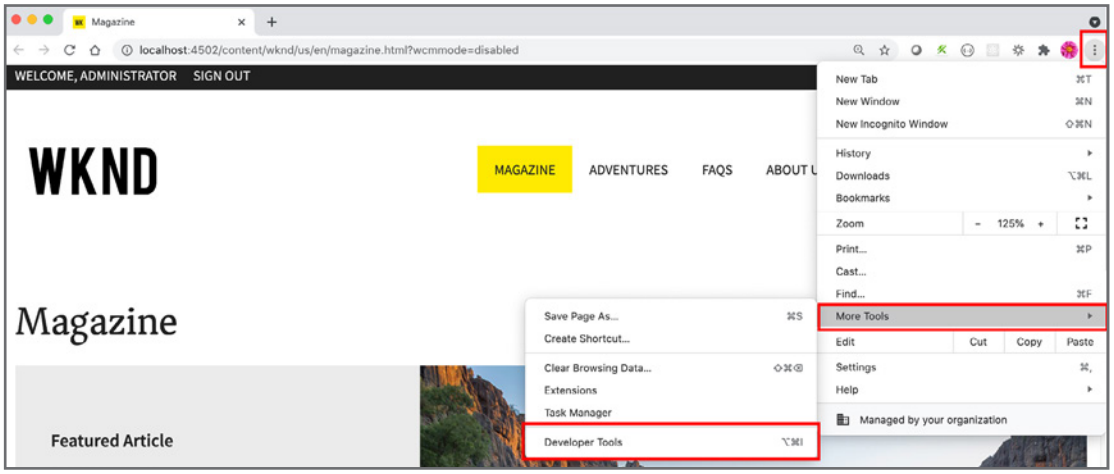
(Optional) Exercise 9: Investigate the Core Component Use of the Adobe Client Data Layer

Modern AEM integrations to Adobe Analytics and Adobe Target are through Adobe Launch. The Adobe Client Data Layer can expose component data to Adobe Launch or any other third party consumption software. In this exercise, you will investigate how the Core Components enable and use the data layer.

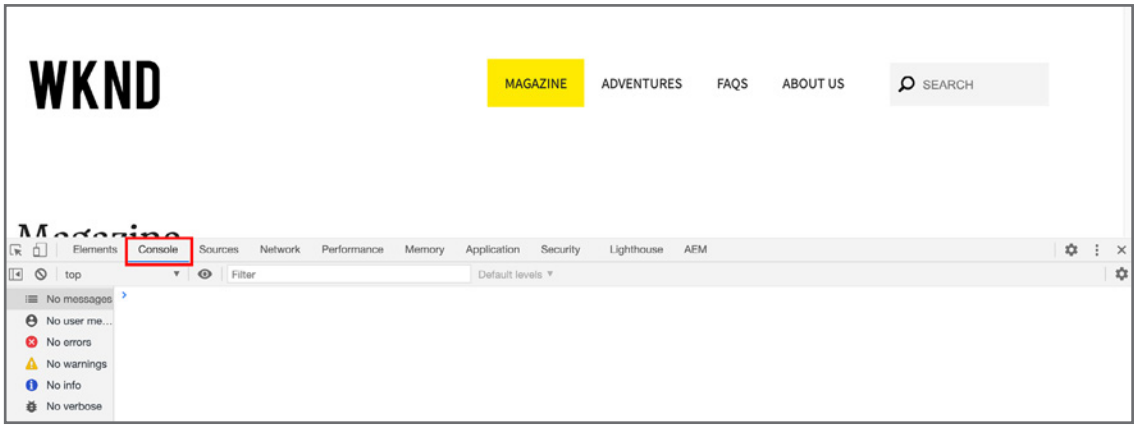
Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service
- WKND project installed

1. In your browser, open the AEM homepage and navigate to **Sites** > **WKND** > **United States** > **WKND Adventures and Travel**.
2. Select the **Magazine** page and open the page editor.
3. Click the **Page Information menu** dropdown and select **View as Published**. This opens a new tab with the URL http://localhost:4502/content/wknd/us/en/magazine.html.
4. Access your browser's developer console. This is typically found in the developer tools of a modern browser.



5. In the developer tools, select the Console tab.

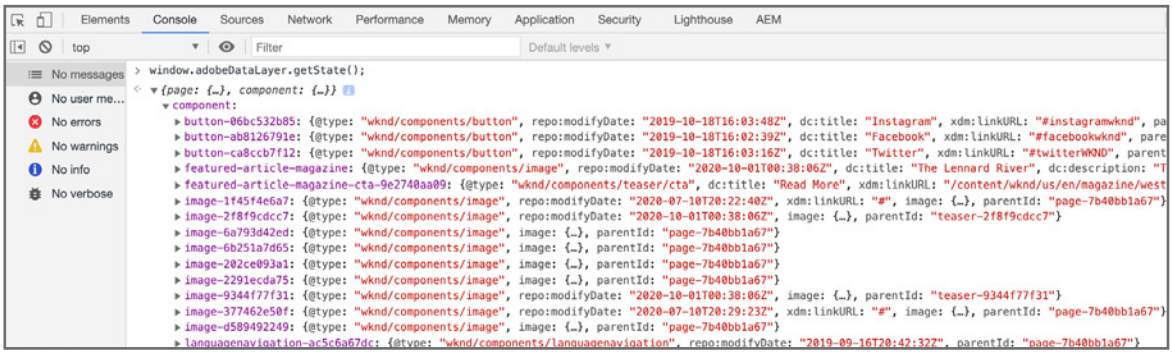


6. Issue the following command in the Console window:

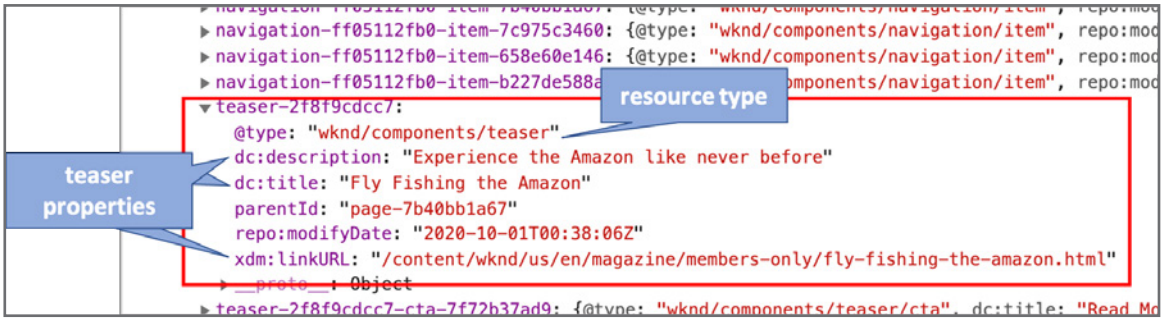
```
window.adobeDataLayer.getState();
```

7. Expand the **{page: {...}, component: {...}** response line.

8. Expand **component** to see all components putting data into the data layer.



9. Expand the first **teaser** element under components to see the data that the teaser component has put into the data layer.



10. Expand the first **image** element under components and then expand the **nested image** element to see the data that the image component has put into the data layer.

11. Expand a few of the other component elements to familiarize yourself with the data that the components put into the data layer.

Exercise 10: Enable a Website for the Adobe Client Data Layer

The first step to start using the Adobe Client Data Layer on your website is it needs to be enabled. The archetype has a parameter to automatically configure this for your project.
In this exercise, you will verify that the website has been enabled for the Adobe Client Data Layer.

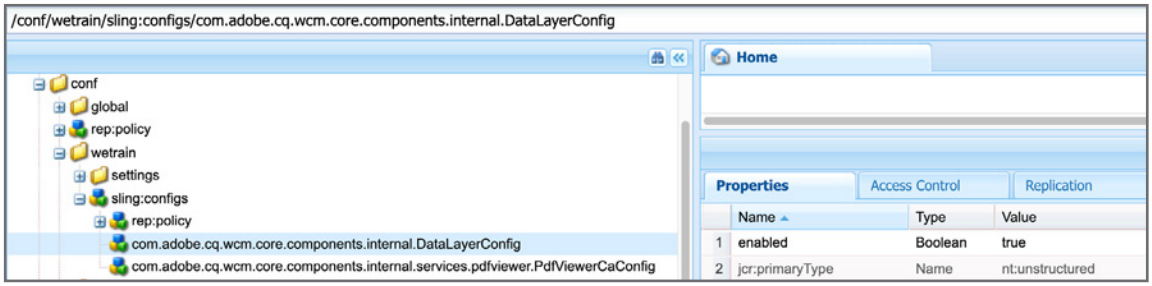
Prerequisites:

- Local author instance of AEM running on port 4502
- we.Train site deployed to the local AEM instance

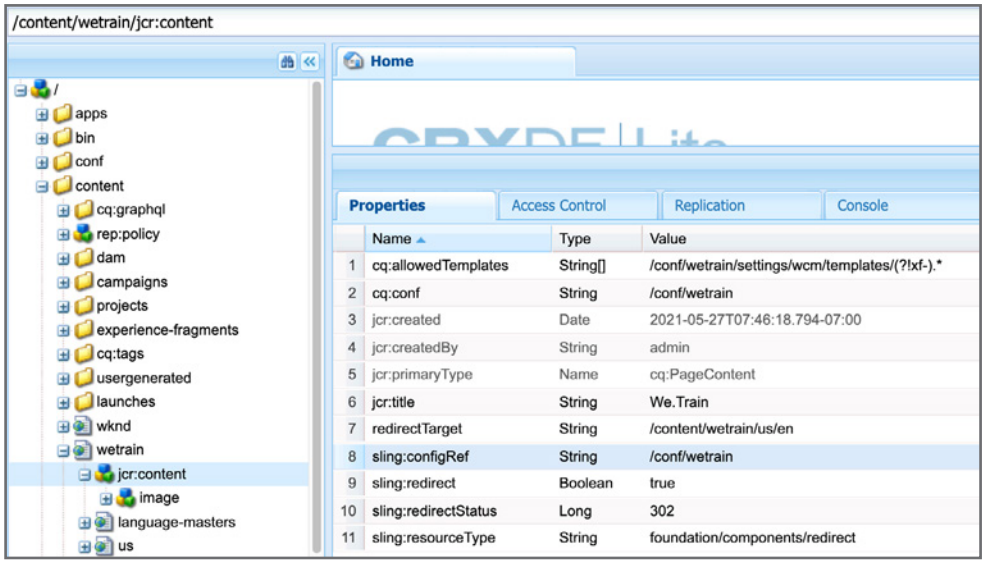
Task 1: Investigate Data Layer Enablement Cofigurations for a Website

The Maven AEM Project Archetype generated the project with the data layer enabled.

1. In **CRXDE Lite**, navigate to **/conf/wetrain**.
2. Expand the **wetrain** node.
3. Select the **com.adobe.cq.wcm.core.components.internal.DataLayerConfig** node. Notice that it contains a property: **enabled = true**.



4. Navigate to **/content/wetrain/jcr:content.** Notice the **jcr:content** node contains a property: **sling:configRef = /conf/wetrain**.



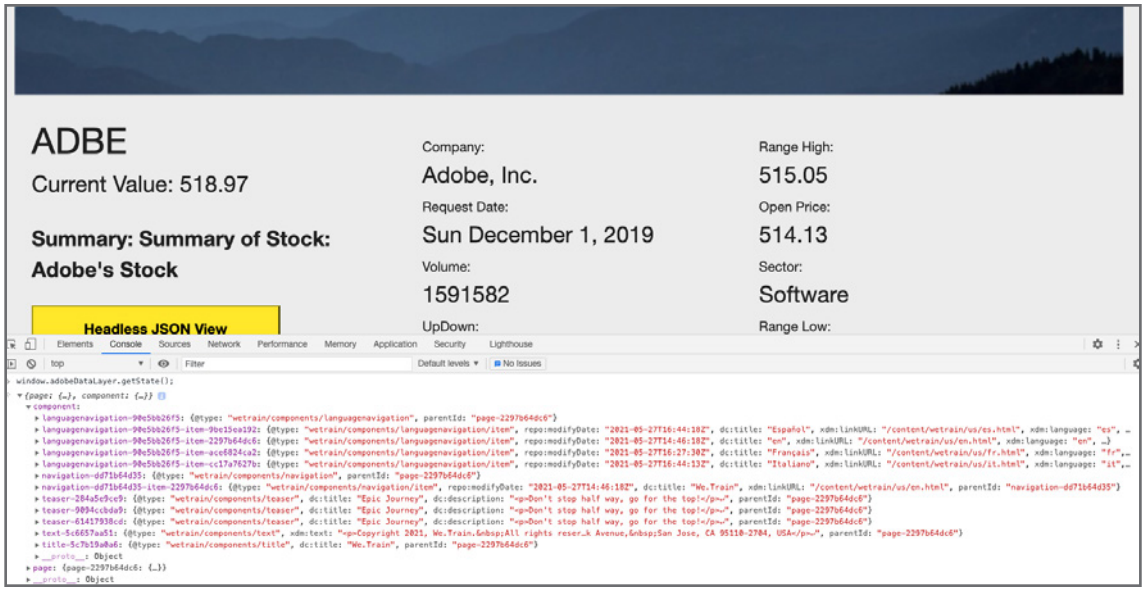
Task 2: Examine the Data Layer Output for the Proxy Components

1. Navigate to the **We-Train site**, select the **en** page and open the **Page Editor.**
2. Using the Page Information menu, **View as Published** to open the page without the Page Editor.
3. Similar to last exercise, open your browsers developer console.
4. In the developer tools Console issue the following command:

```
window.adobeDataLayer.getState();
```

You will notice that the data layer is now enabled for the We.Train site.

5. Expand the **{page: {…}, component: {…}** response line and then expand **component**. You will notice that, even though the Stockplex component is on the page, it is not listed under the component element. The data layer is enabled for the website, however the Stockplex component is not yet enabled to populate the data layer.



Exercise 11: Enable a Custom Component to Populate the Adobe Client Data Layer

The functional requirements require component data be exportable so Adobe Analytics and Adobe Target can consume it easily. Using the Adobe Client Data Layer in a custom component can expose the custom component for analytics and targeting purposes. In this exercise you will implement the Adobe Client Data Layer in your custom component and validate data layer use.

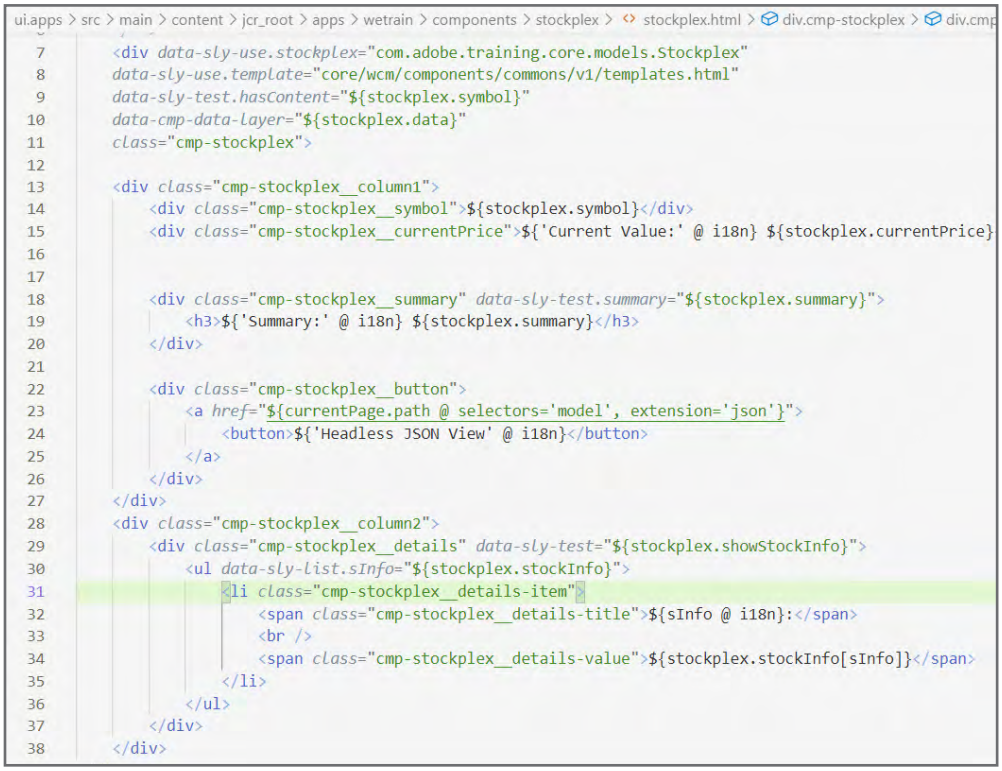
Prerequisites:

- Local author instance of AEM running on port 4502
- wetrain AEM project
- Existing stockplex component

Task 1: Update the Script to Use the Data Layer

1. In your IDE, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/stockplex**.
2. Open the **stockplex.html** file.
3. In your Exercise_Files-DWC, navigate to **ui.apps/src/main/content/jcr_root/apps/ wetrain / components/stockplex**.

4. Copy the contents of **datalayer_stockplex.html** and paste to replace the contents of your **stockplex.html** file.



5. Save the changes.

6. Examine the script to understand how the data layer was enabled for the Stockplex component (see line 10). This one line of HTL allows the getData() method to be exported to the data layer. The getData() method is in the StockplexImpl.java model in the core model.

Task 2: Validate Data Layer Use

1. In your IDE, open a terminal window.

Note: You must deploy to AEM. Even though the IDE has synchronized the script, you must build the project to deploy the OSGI bundle containing the Sling Model.

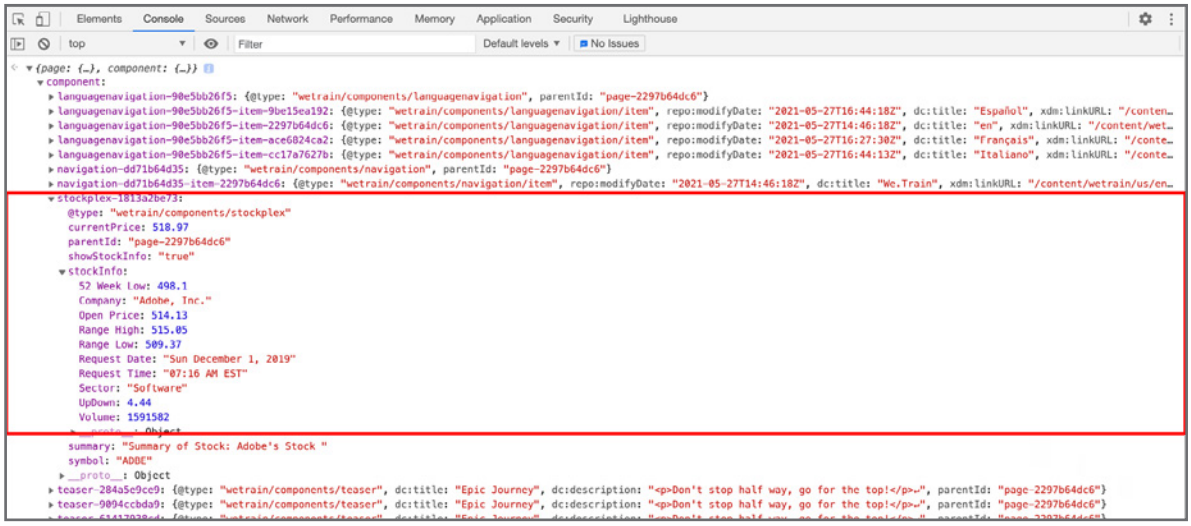
2. Type in the following command to deploy your project to AEM:

```
mvn clean install -PautoInstallSinglePackage
```

3. In your browser, open the AEM homepage and navigate to the **We.Train site**.
4. Select the **en** page and open the **Page Editor.**
5. Using the Page Information menu, **View as Published** to open the page without the Page Editor.
6. Similar to previous exercises, open your browser's developer console.
7. In the developer tools Console, issue the following command:

```
window.adobeDataLayer.getState();
```

8. Expand the **{page: {...}, component: {...}** response line and then expand **component**.



(Optional) Exercise 12: Enable a Clickable Event in the Adobe Client Data Layer

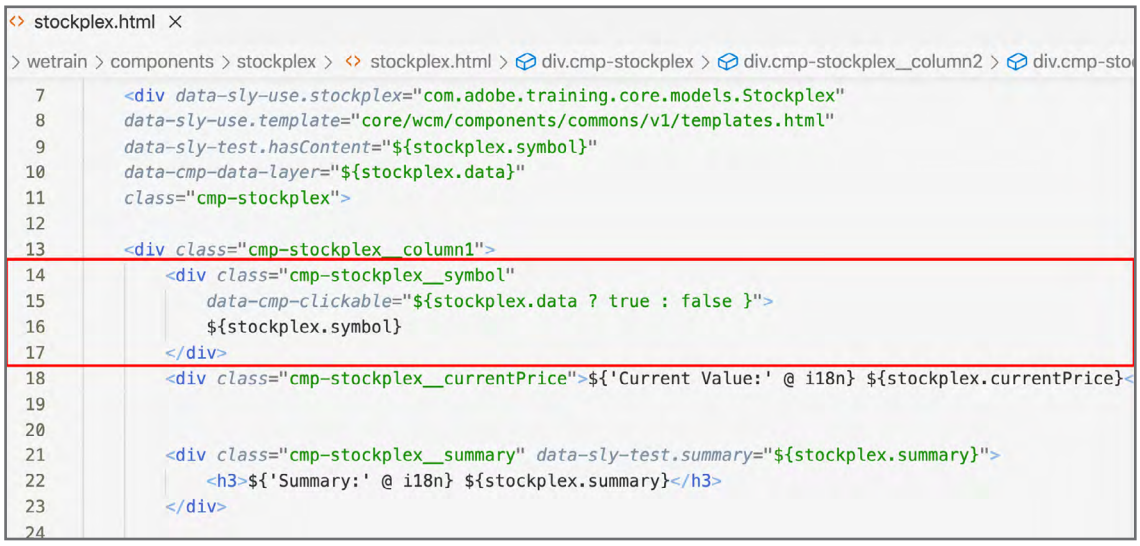
Going beyond functional requirements, as a developer you may want to track event clicks so they can be sent to Adobe Launch for analysis. In this exercise you will update your HTL to produce an event and handle it with JavaScript.

Prerequisites:

- Local author instance of AEM running on port 4502
- We.train AEM project
- Existing stockplex component

Task 1: Update the Script to Add a Click Event

1. In your IDE, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components**.
2. Open **stockplex.html.**
3. In the Exercise_Files-DWC folder, navigate to **ui.apps/src/main/content/jcr_root/apps/wetrain/components/stockplex**.
4. Copy contents of the **clickable_stockplex.html** file and **paste** it into the **stockplex.html** file in your IDE.



5. Save the changes.
6. Examine the script to see how the Stockplex component script enables the clickable event.

Note: If your IDE has auto-sync on save enabled, the stockplex.html file has already imported into AEM. If you are not using this feature, you will need to run maven install:

```
mvn clean install -PautoInstallSinglePackage
```

Task 2: Test the Click Event

1. In a browser tab, navigate to **We.Train** > **Language Masters**. Open the **en** page in the page editor.
2. Using the Page Information menu, click **View as Published** to open the page without the editor.
3. Similar to previous exercises, open your browser's developer console.

To capture the click event in our browser, we need to manually add some JavaScript using the developer tools console.

4. In your Exercise_Files-DWC, go to **training-files/custom-cmps** folder and open the **javascript-event-listener.txt** file.

```
function stockplexClickHandler(event) {
    var dataObject = window.adobeDataLayer.getState(event.eventInfo.path);
    if (dataObject != null && dataObject['@type'] === 'we-train/components/stockplex') {
        console.log("Stockplex Clicked!");
        console.log("Stockplex symbol: " + dataObject['symbol']);
    }
}
window.adobeDataLayer.push(function (dl) {
     dl.addEventListener("cmp:click", stockplexClickHandler);
});
```

5. Copy the contents of **javascript-event-listener.txt** and **paste** into the developer tools console and click the **Return/Enter** key.

Caution! Do not refresh the browser during this exercise. If you refresh you will lose the JavaScript.

6. Click the ADBE stock symbol on the page and you will notice that the click event has been captured.

(Optional) Exercise 13: Examine Data Layer Enablement in the Sling Model for a Component

Earlier, when you implemented the Hero component, you created a Sling Model for the component. That Sling Model extended the Image Component Sling Model. Take a moment and investigate the similarities and differences in the implementation of the getData() method in the Sling Models for the Hero and Stockplex components.

In your <AEM Project>, navigate to **core/src/main/java.com/adobe/training/core/models/**.

You will find the code for the Hero Component and Stockplex Component in **Hero.java** and **/impl/StockplexImpl.java** (respectively).

References

- Adobe Client Data Layer
- Core Component Events
- Integrating with Analytics and Target
- Customizing Core Components
- Coding for the Style System
- Configuring a Design Dialog
- Sling Models

Work with Page Templates

Introduction

In Adobe Experience Manager (AEM), template authors use templates for creating pages. Authors can create and configure the editable templates without a development project or iteration. However, to enable this capability, developers must set up the base templates configurations and create client libraries and components to be used in the templates.

Objectives

After completing this module, you will be able to:

- Explain templates for pages
- Create editable templates
- Create pages from editable templates
- Build your website
- Include header and footer fragments

Templates

Content authors can easily create pages and content from the AEM Sites console.

Templates defines the content structure with JCR nodes and properties. Templates use the node type **cq:Template** as the root node.

Editable Templates

- Are created and edited by authors
- Help define the structure, initial content, and content policies for pages
- Maintain a dynamic connection between the template and pages
- Use content policies (edited from the template editor) to persist the design properties
- Are stored under /conf

Templates Console

The Templates console enables template authors to:

- Create a new template
- Edit the template by using the template editor
- Manage the template life cycle

You can access the Templates console from the **Tools** > **General** section.

Template Editor

The template editor enables template authors to:

- Add the available components to the template and position them on a responsive grid
- Preconfigure components
- Define the components that you can edit on the resultant pages (created from the template)
- Compose templates out of available components
- Manage the lifecycle of templates

Similar to the Page Editor modes, the Template Editor has several unique modes:

- Structure: The structure enables template authors to define the components, such as header, footer, and layout container for a page. Template authors can add a paragraph system to the template, which helps page authors to add and remove the components on the resultant pages. When components are locked, page authors cannot edit the content. In Structure mode, any component that is the parent of an unlocked component cannot be moved, cut, or deleted.
- Initial Content: Template authors can define the content that needs to be included in all pages, by default. When a component is unlocked, template authors can define the initial content that will be copied to the resultant pages created from the template. Page authors can edit these unlocked components on the resultant pages. In the initial content mode (and on the resultant pages), you can delete any unlocked components with an accessible parent, such as components within a layout container.
- Layout: The Layout enables template authors to predefine the template layout for the required device formats.

Content Policies

You can define content policies for templates and components from the template editor.
The content policies:

- Connect the predefined page policies to a page. These page policies define various design configurations.
- Enable you to assign a predefined design configuration to selected components. This helps preconfigure a component's behavior on the resultant page.
- Enable you to add the allowed components and default components to the template.
- Define the number of columns (responsive settings) in the template.

Creating Editable Templates

Below are common steps to use editable templates:

1. Create a template folder using context-aware configurations.
2. Create a template-type for your template.
3. Create a new template from the template type.
4. Define additional properties for the template, if required.
5. Edit the template to define the structure, initial content, layout, and content policies.
6. Enable the template.
7. Publish the template so that it is available on the publish environment.
8. Make the template available for the required page or the branch of your website.

Creating the Template Folder

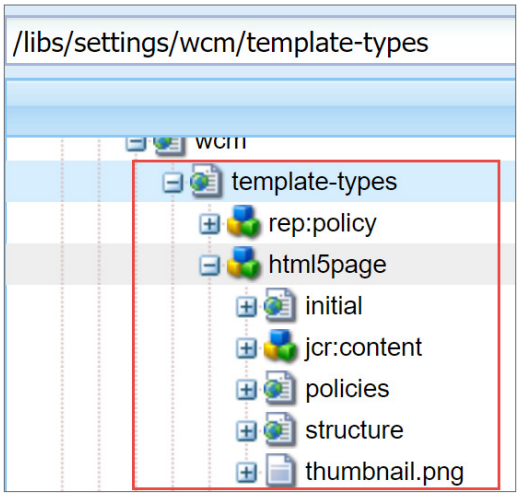
A new template folder can be auto generated by using the Configuration Browser. You can access the **Configuration Browser** console from the **Tools** console. You can create a multi-tenant environment where tenants can have different editable templates, configurations, cloud services, and data models.

Creating a Template Type

When creating a new template, you must specify a template type. The template type is copied to create the template, and the structure and initial content of the selected template type is used to create the new template. After the template type is copied, the only connection between the template and the template type is a static reference for information purposes.
Template types help you define the:

- Resource type of the page component
- Policy of the root node, which defines the components allowed in the template editor

The out-of-the box template-types are stored under **/libs/settings/wcm/template-types,** as shown:



Note: A project generated from the archetype will already have a template type for you to use.

Template Definitions

The definitions for editable templates are stored under the tenant specific folder. For example:

- /conf/<my-folder>/settings/wcm/templates
- /conf/<my-folder-01>/<my-folder-02>/settings/wcm/templates

The root node of the template is of type **cq:Template** with the following skeleton structure:

```
<template-name>
  initial
    jcr:content
      root
        <component>
        ...
        <component>
  jcr:content
    @property status
  policies
    jcr:content
      root
        @property cq:policy
        <component>
          @property cq:policy
        ...
        <component>
          @property cq:policy
```

The main elements are:

- <template-name>
- jcr:content
- structure
- initial
- policies
- thumbnail.png

jcr:content node hierarchy

The jcr:content node holds the following properties of the template:

- Name: jcr:title
- Name: status
  - Type: String
  - Value: draft, enabled, or disabled

Structure Node Hierarchy

The structure node defines the structure of the resultant page. The structure of the template is merged with the initial content (/initial) when creating a new page. Any changes to the structure are reflected on any pages created with the template.

The root (structure/jcr:content/root) node defines the list of components that are available in the resulting page. The components defined in the template structure cannot be moved or deleted from any resultant pages. After a component is unlocked, the editable property is set to true. After a component that already contains the content is unlocked, the content is moved to the initial branch. The **cq:responsive** node holds the definitions for the responsive layout.

Initial Node Hierarchy

The initial node defines the initial content that a new page has upon creation. The initial node contains a **jcr:content** node that is copied to any new pages and is merged with the structure (/structure) when a new page is created. Any existing pages are not updated if the initial content is changed after creation. The root node holds a list of components to define what is available in the resulting page. If the content is added to a component in Structure mode and that the component is subsequently unlocked (or vice versa), the content is used as initial content.

Policies Node Hierarchy

The content (or design) policies define the design properties of a component. For example, the components available or the minimum/maximum dimensions. These are applicable to the template (and pages created with the template). Content policies can be created and selected in the template editor. The property `cq:policy`, on the **/conf/<your-folder>/settings/wcm/templates/<your-template>/policies/jcr:content/root** root node, provides a relative reference to the content policy for the page's paragraph system. The property **cq:policy**, on the component-explicit nodes under root, provides the links to the policies for the individual components. The actual policy definitions are stored under **/conf/<your-folder>/settings/wcm/policies/wcm/foundation/components**.

Note: The paths of policy definitions depend on the path of the component. cq:policy holds a relative reference to the configuration itself.

Editing the Template

After creating a template, you can edit the template from the template editor. The changes made to the template impact the existing pages depending on the template editor modes. For example, if you make changes to the structure of the template, it applies immediately to all pages created from the template. It is also possible to define the initial content for a template, which is copied over to newly created pages.

Enabling the Template

Before you use a template, you must enable from the templates Console.
Once enabled, to use the template in a website or branch of a website. Define the **Allowed Template paths** on the **Page Properties** of the appropriate page or root of the page of a sub-branch.

Publishing the Template

As the template is referenced when a page is rendered, the fully configured template must be published from the **Templates** console to make it available on the publish environment.

Exercise 1: Create and Investigate a Page Template

You have created a Maven project based on the latest archetype which included a configured template-type already. In this exercise you will use this generated template-type to create and explore an editable template.

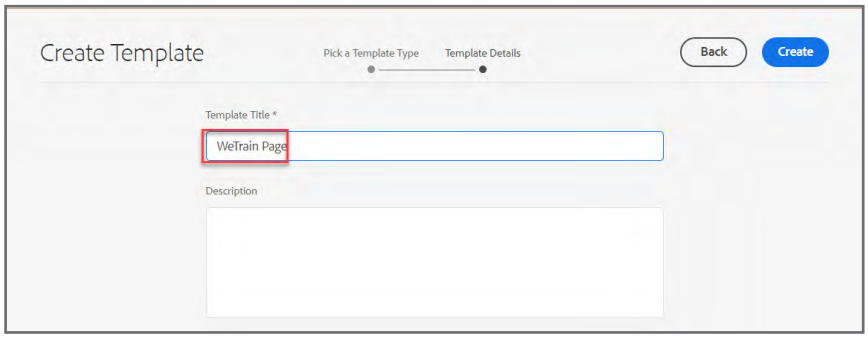
Prerequisites:

- Local author service running
- A Maven project imported into your IDE and installed on the author service

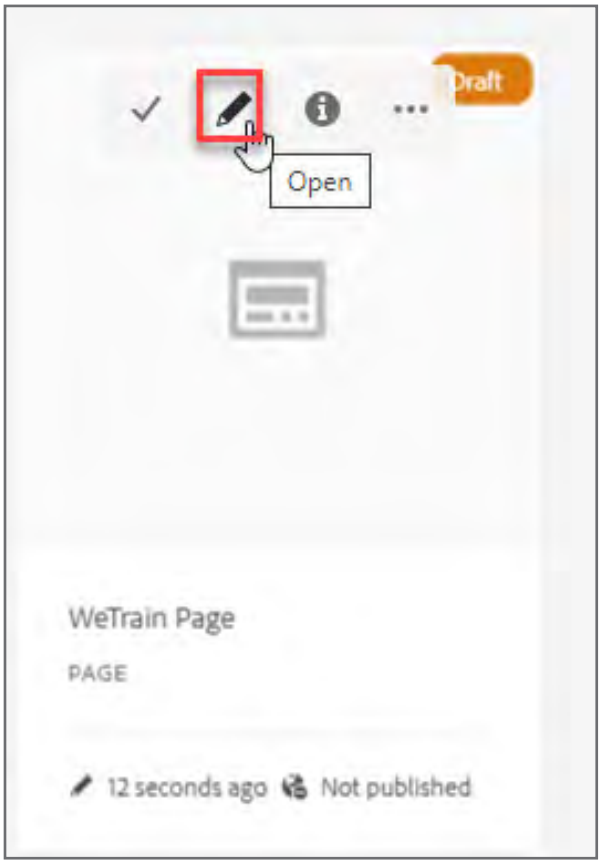
1. Navigate to **Adobe Experience Manager** > **Tools** > **Templates,** if you are not in the **Templates** console already.
2. Click the **We.Train** folder and click **Create** from the actions bar to create a new Editable template. The **Create Template page** opens. Notice that template types are already available.
3. On the **Create Template** page, select the **Page** Template Type, as shown:

This template type is installed from the Maven project generated from the archetype.

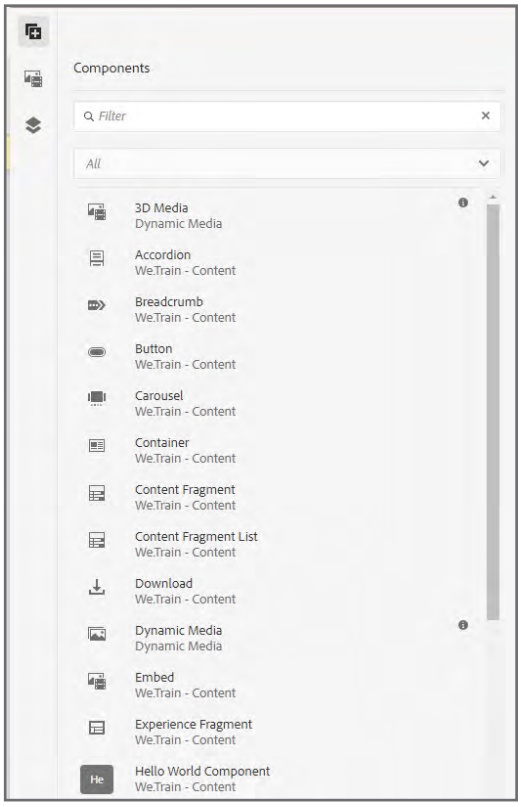
4. Click **Next.** The **Create Template** page opens.
5. In the **Template Title** box, enter **WeTrain Page**, as shown:



6. Click **Create**. The **Success** dialog box opens.
7. Click **Done** in the **Success** dialog box.
   The resulting page shows our newly created template in **Draft** mode.
8. Hover the cursor over the **WeTrain Page** template and click the **Open** icon, as shown. The **WeTrain Page** template opens on a new browser tab for editing.

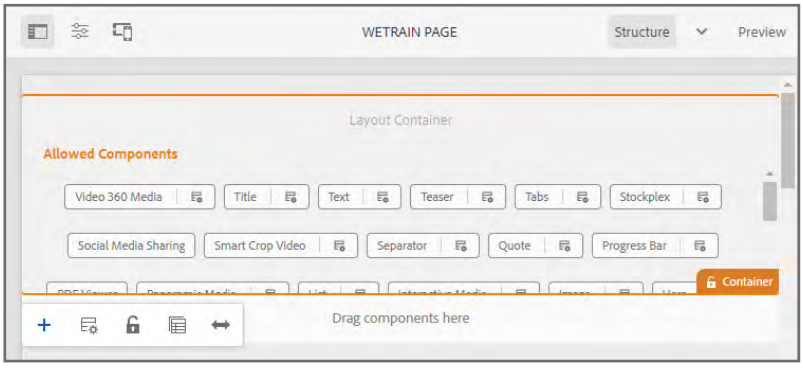


9. Notice it opens in **STRUCTURE** mode.
10. Expand the side panel, if it is not expanded already.



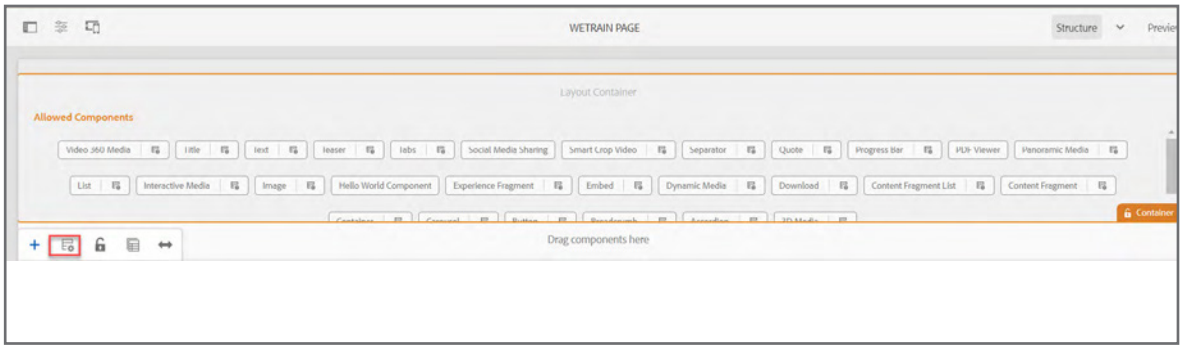
Note: The list of components shown here are the components that you can use as you edit the template structure, and the components enabled in the container which the page authors can use on pages built from this template.

The **Container [Root]** allows template editors to add components to the structure of the template.



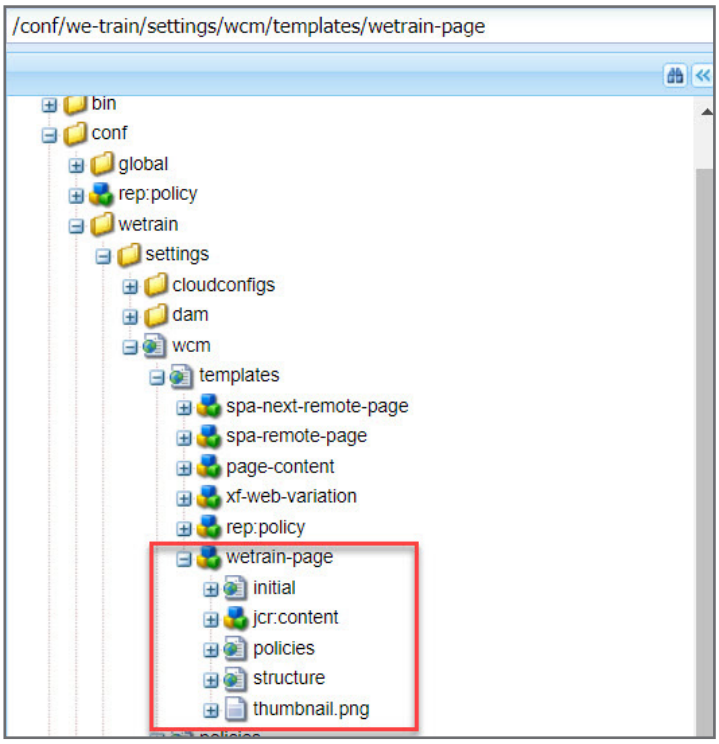
Note: The root container is visible to template authors only when editing the template. This container is not on the page.

11. The template already has a Container component enabled with allowed components. This was defined in the **Page** template-type installed from our Maven project:

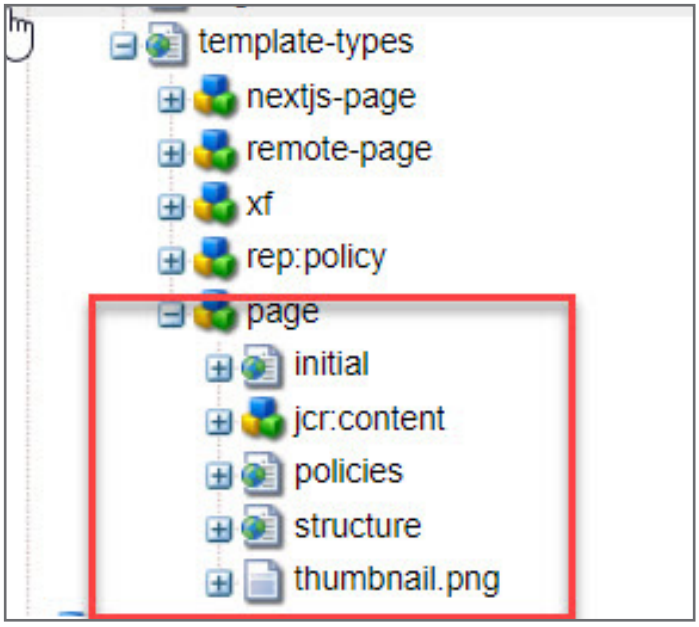


To verify this:

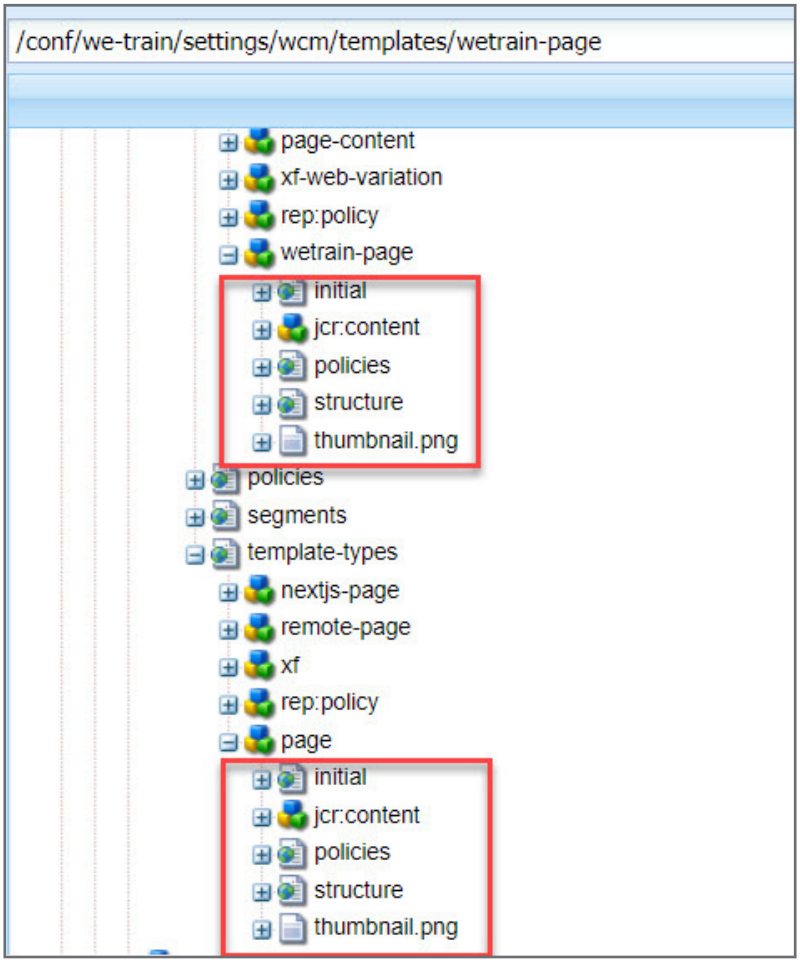
12. Click **Adobe Experience Manager** from the header bar. The **Tools** console opens.
13. Click **CRXDE Lite**. The **CRXDE Lite** page opens.
14. Navigate to the **/conf/we-train/settings/wcm/templates/wetrain-page** node.



15. Similarly, navigate to the **/conf/we-train/settings/wcm/templates-types/page** node:

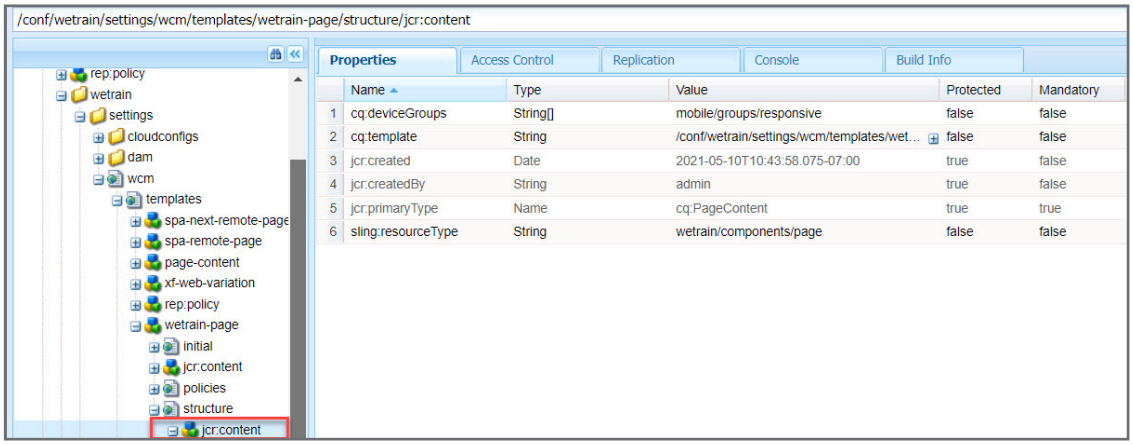


16. Notice how the sub nodes of these two locations mirror each other.



17. Navigate to **/conf/we-train/settings/wcm/templates/wetrain-page** and expand the **structure** node.

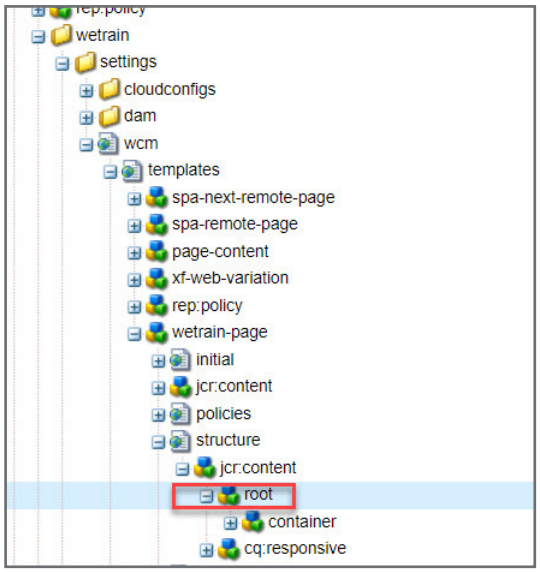
18. Select its **jcr:content** child node to view the Properties, as shown:



19. Notice the **sling:resourceType** property which directs Sling rendering to your project's proxy page component.

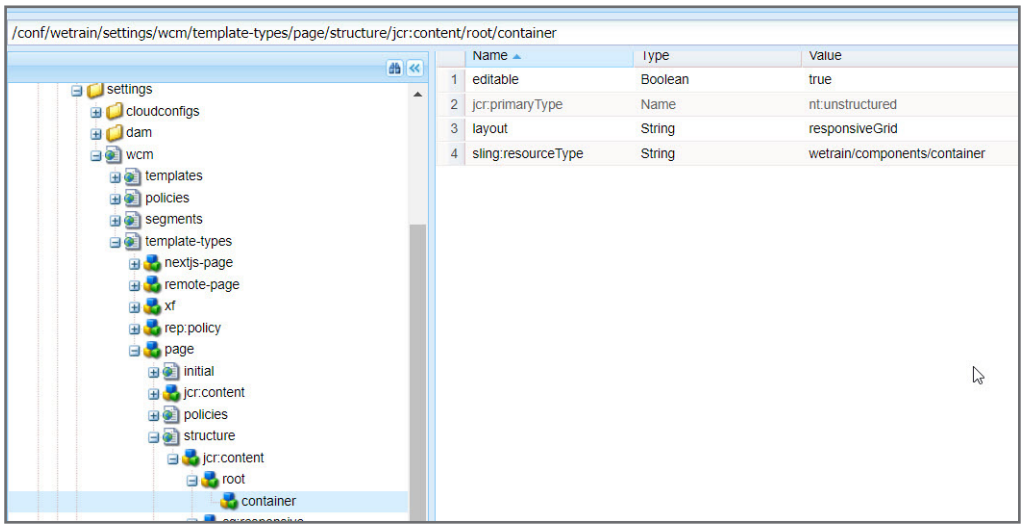
20. Notice the same setting exists in the **/conf/we-train/settings/wcm/templates/wetrain-page/initial/jcr:content** node.

21. Expand the root node under **/conf/we-train/settings/wcm/templates/wetrain-page/structure/jcr:content.**

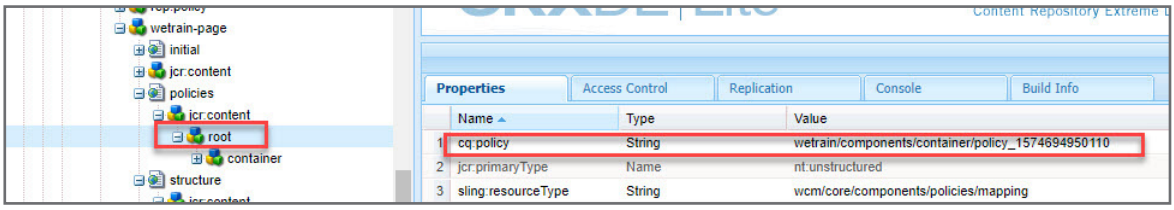


Note: "root" is the root container component of the template. "container" is the container component added to the template for page authors to add content on pages built from the template. This structure was created by the template type.

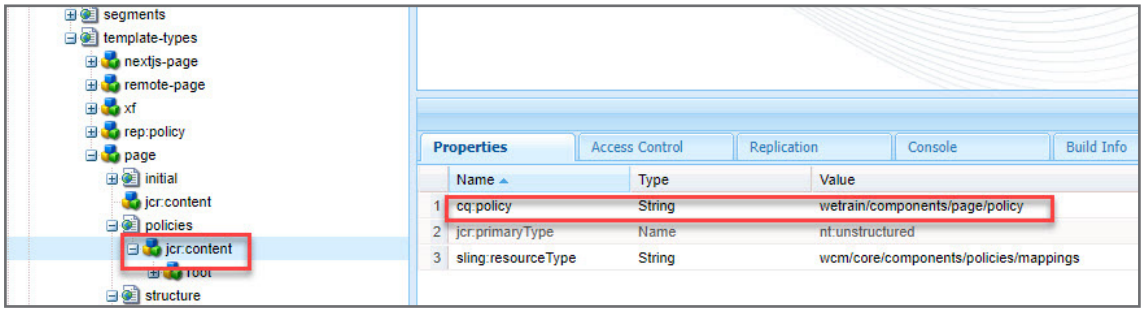
22. Expand **..template-types/page/structure/jcr:content/root** and view some of the node properties, as shown:



23. The same principle holds true for the policies you observed on the **WeTrain** page template.
    a. Select **../templates/wetrain-page/policies/jcr:content/root** and note the **cq:policy** property & its unique ID:



    b. Select **../template-types/page/policies/jcr:content/root** and note the same unique ID in its **cq:policy**:

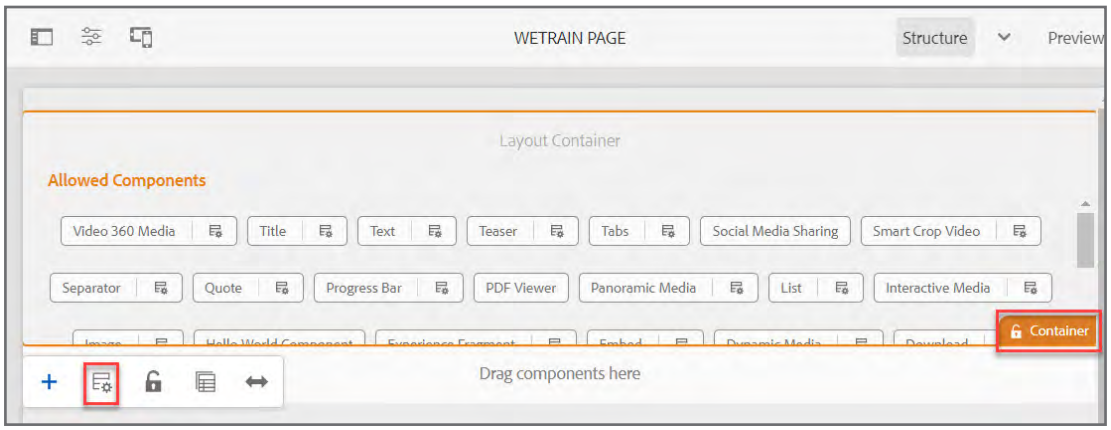


With policies, the actual information about what the policy contains is stored global to the given "context", so it can be used by other templates or template types of the context. For ex. **/conf/<context>/settings/wcm/policies/<context>/components**

24. In this use case, it is: **/conf/we-train/settings/wcm/policies/we-train/components/container:**

Other key policies provided by the template type in the WeTrain page template are some asset to component mappings which are set in the container.

25. Return to the browser tab with the **WeTrain** Page template.
    a. Select the unlocked Container with **Allowed Components** and click the Policy icon, as shown:

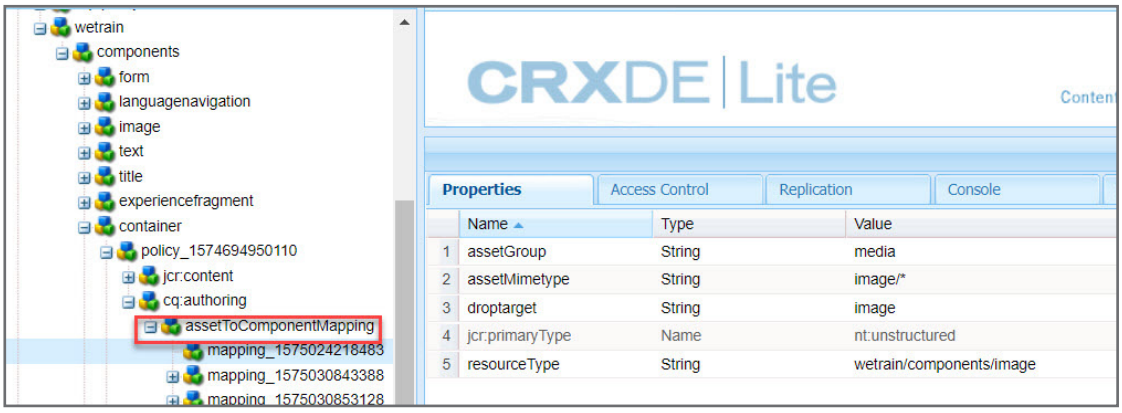


The **Container** page opens.

    b. On the right-hand side, under **Properties** column, select the **Default Component** tab and notice the three entries for Image, Experience Fragment, and Content Fragment components.
    c. Each entry indicates what Mime type(s) are accepted and what Component is applied.

In the event an author directly drops that given asset type into this container, it is a convenience mechanism for adding assets to a container.

26. Back in **CRXDE Lite**, reinspect the policy that was originally set on the container in the template type. Navigate to **/conf/we-train/settings/wcm/policies/we-train/components/container/policy_<uniqueID>/cq:authoring/assetToComponentMapping** and examine the mappings:



Note: Your policy numeric ID will be different.

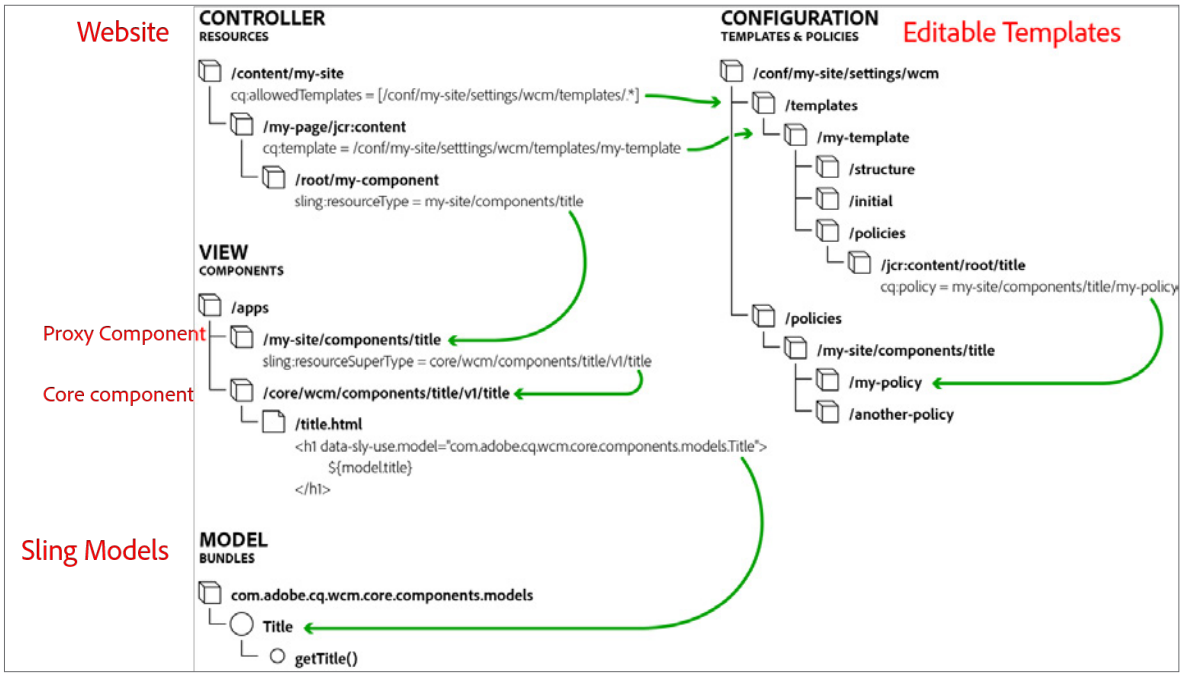
The above examples holds true for the other child nodes of the template type. The properties that you set in the template type's child nodes are written into the created template.

Creating Pages from Editable Templates

In the **Sites** console, you can create the pages of your website by using a template. The pages created from editable templates:

- Are created with a subtree that is merged from the structure and initial in the template.
- Have references to the information held in the template and the template type. This is achieved with a jcr:content node with the following properties:
  - cq:template: Provides the dynamic reference to the actual template, and enables changes to the template to be reflected on the actual pages.
  - cq:templateType: Provides a reference to the template type.

The below diagram shows how templates, content, and components interrelate:



In the above diagram:

- Controller: The resultant page that references the template (**/content/<my-site>/<my-page>)**

    - The content controls the entire process. According to the definitions, it accesses the appropriate template and components.
- Configuration: The template and related content policies define the page configuration (/**conf/<my-folder>/settings/wcm/templates/<my-template>**)
- Model: The OSGI bundles implement the functionality.
- View: On both author environment and publish environment, the content is rendered by components (/apps/<my-site>/components).

When rendering a page:

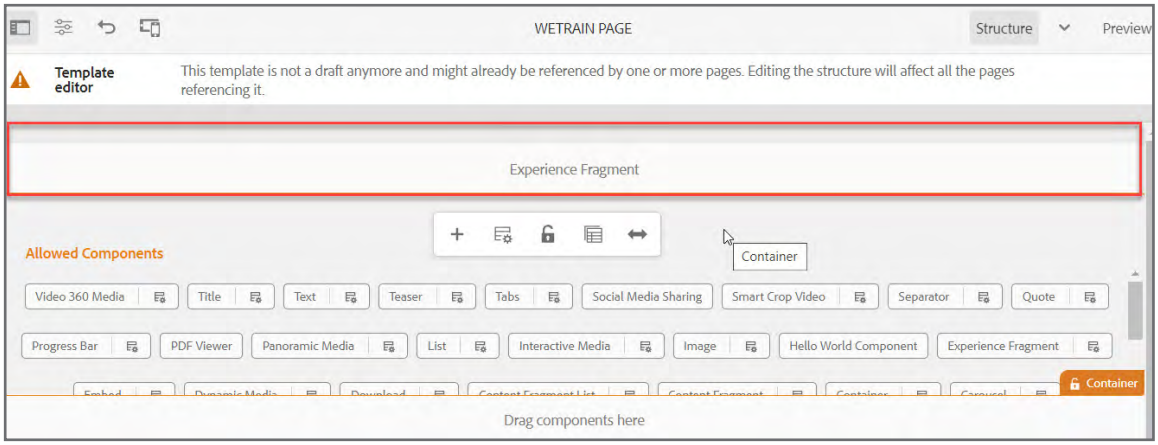
- The **cq:template** property of its **jcr:content** node is referenced to access the template that corresponds to that page.
- The page component will merge the **structure/jcr:content** tree of the template with the **jcr:content** tree of the page.
- The page component will allow the author to edit only those nodes of the template structure that are flagged as editable.
- The relative path of the component is taken from the jcr:content node when rendering a component on a page.
- The **cq:policy** property of the component:
    - Points to the actual content policy (holds the design configuration for that component).
    - Helps you have multiple templates that re-use the same content policy configurations.

Exercise 2: Finish Creating Initial Templates

The generated template as well as the one created earlier in this module are minimally created for testing purposes. In this exercise, you will perform several tasks to make these templates ready for production.

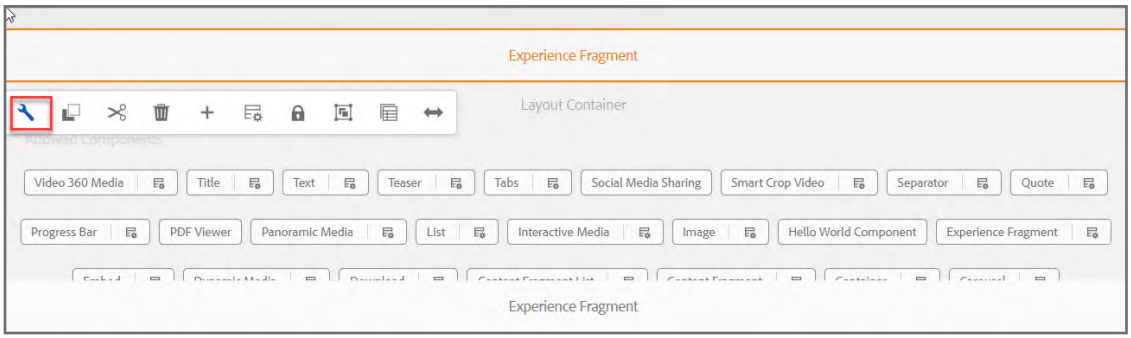
Task 1: Add a Header and Footer Using Experience Fragments

1. Hover over the **WeTrain Page** template and select the Edit icon (pencil image). The **WeTrain Page** template opens in **Structure** mode.
2. Drag and drop an **Experience Fragment** component above the Layout Container with Allowed Components, as shown:

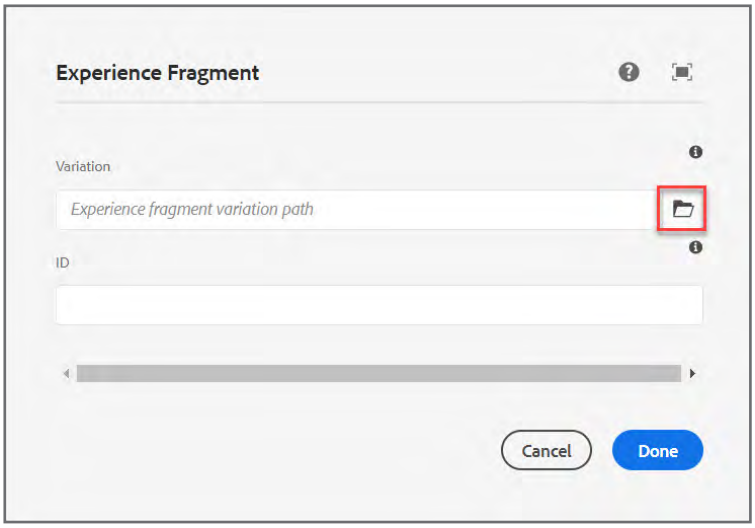


3. Add another **Experience Fragment** component below the Layout Container with Allowed Components.

4. Select the Wrench icon on the top **Experience Fragment** component:

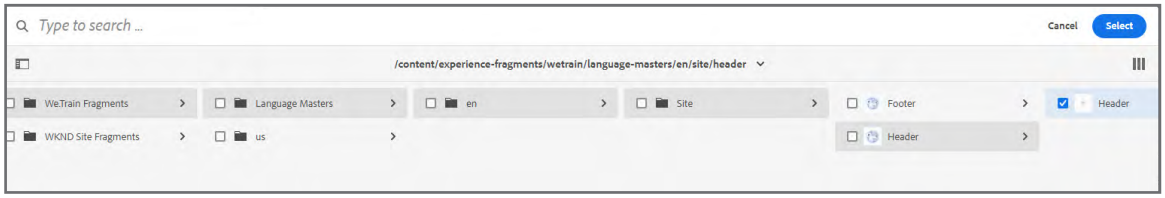


5. Configure the field Variation by clicking the Open Selection dialog, as shown:

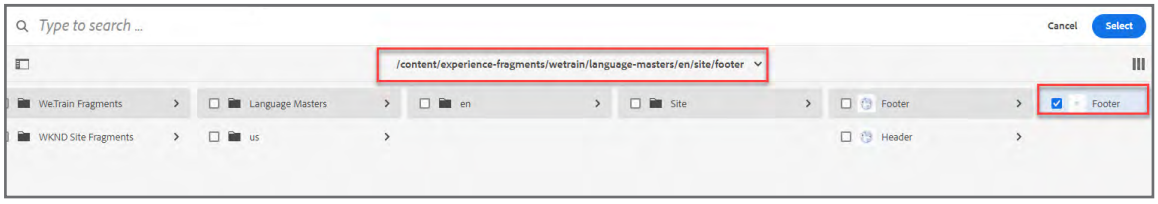


The **experience-fragments** window is displayed.

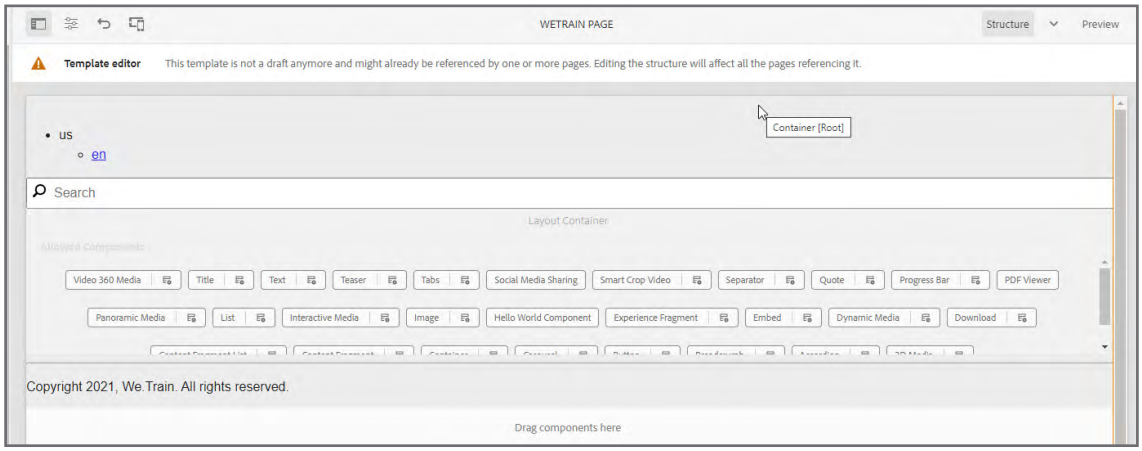
6. Browse to **/content/experience-fragments/wetrain/language-masters/en/site/Header** and select the **Header** Experience Fragment, as shown:



7. Click **Done**.
8. Select the Wrench icon on the bottom **Experience Fragment** component and select the **Footer** Experience Fragment, as shown:

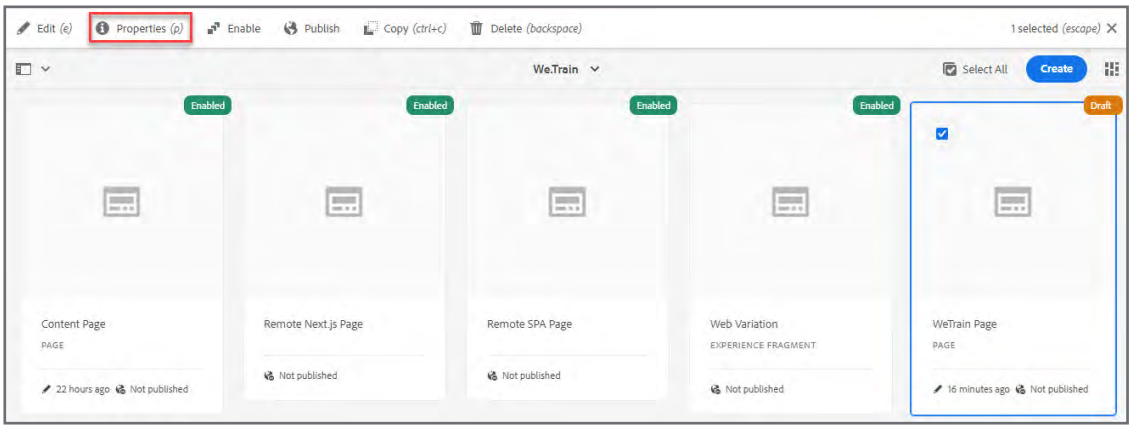


9. Click **Done** to save the changes.
10. Verify the template:



Task 2: Finish and Enable Templates for Page Creation

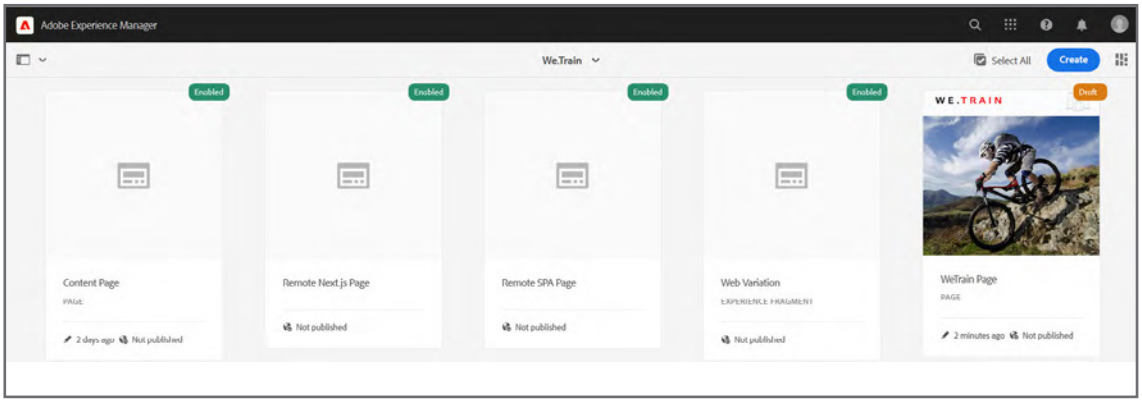
1. Navigate to **Adobe Experience Manager** > **Tools** > **Templates** if you are not in the **Templates** console already.
2. Click the **We.Train** folder and select the **WeTrain Page**.
3. Click **Properties** (p), as shown:



The **WeTrain Page** opens.

4. Click **Upload Image**, as shown:

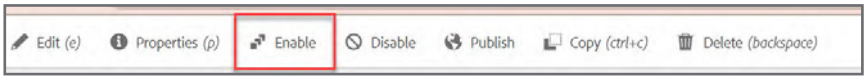
5. In the Exercise_Files-DWC folder provided to you, navigate to the **training files** > **editable templates** folder and select **We.Train-thumbnail-summer.png**
6. Click **Save & Close.**
7. The image is added, as shown:



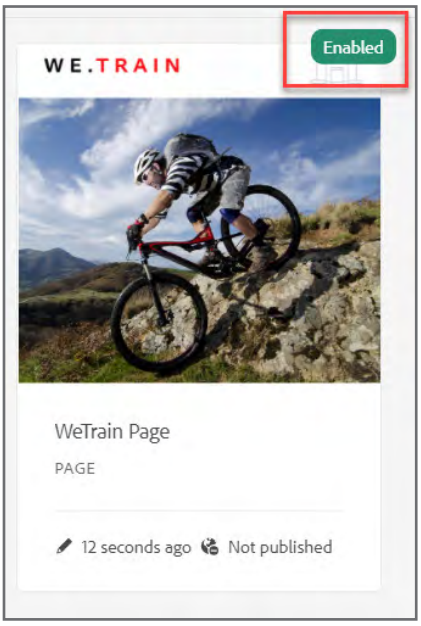
Note: You may need to load the browser's Developer tools and select "Disable cache" in the Network tab to see these changes immediately. This image is added to the visual display of the Template in the Templates console and will be visible in the Sites console for pages built from this template.

8. Hover over the **WeTrain Page** template and select the checkmark icon.

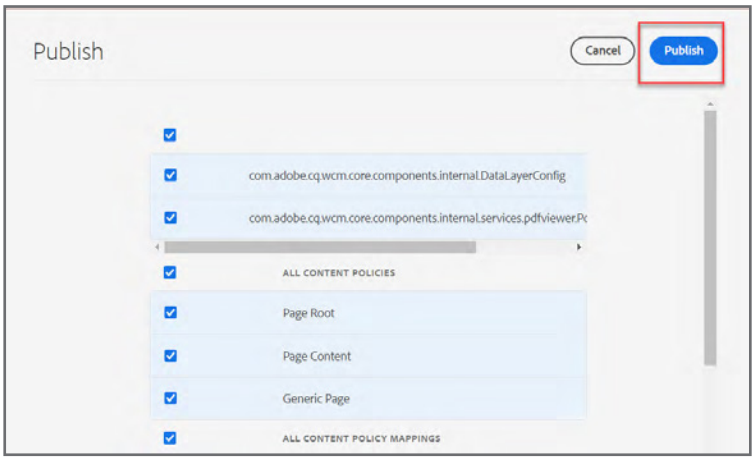
9. In the Actions bar that appears at the top of the browser, click **Enable**, as shown:



10. Click **Enable** to confirm. The template status changes to **Enabled**, as shown:



11. Hover over the **WeTrain Page** template and select the checkmark icon.
12. Select **Publish** in the Actions bar. The **Publish** page opens.
13. Click **Publish** again, as shown:



14. Repeat this entire task for the generated **Content Page** template. In the Exercise_Files-DWC folder provided to you, navigate to the **training files** > **editable templates** folder and select **We.Train-thumbnail-winter.png**

After these steps are performed, you will have two completed templates enabled and published.

Exercise 3: Build Out the Website

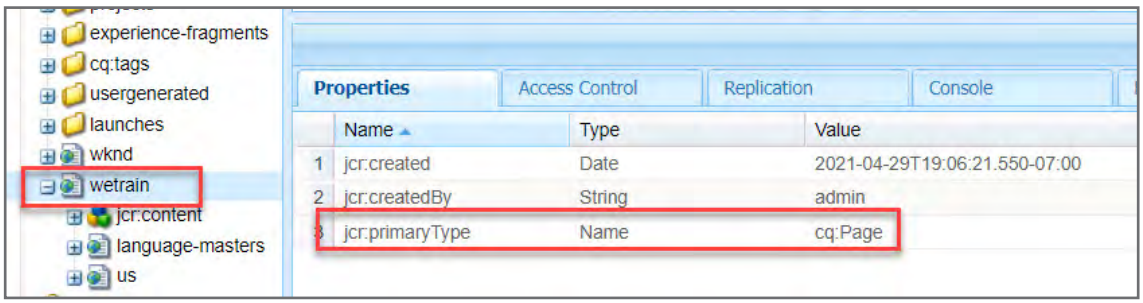
In this exercise you will explore page nodes as well as create a website structure based on the defined templates from earlier in this module.

Task 1: Investigate Different Page Nodes

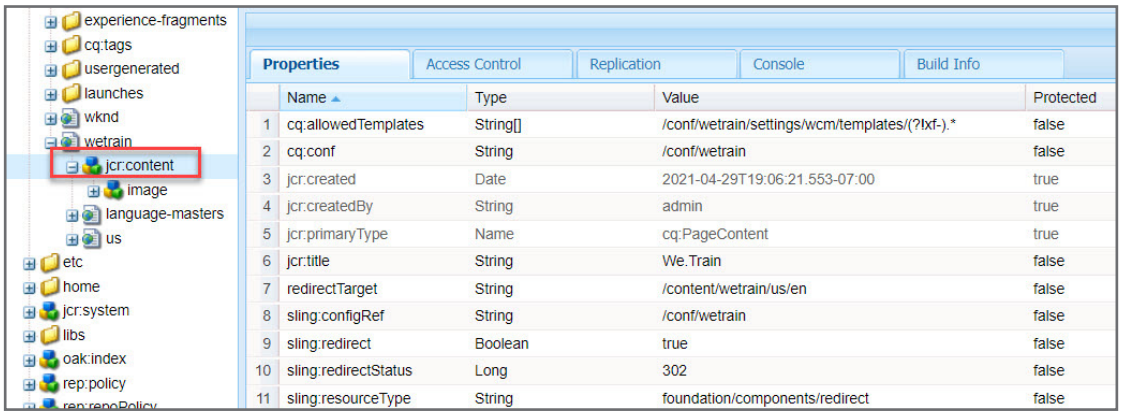
1. In a browser open AEM home page, click **Adobe Experience Manager**.
2. Click the **Navigation** icon on the left panel and click **Sites**. The **Sites** console opens.
3. Notice the three Website roots that you currently have: WKND Site, Campaigns, and We.Train, as shown:

To view the properties in JCR:

4. Refresh **CRXDE lite**, navigate to the **/content/we-train** node and observe the properties. Notice the jcr:PrimaryType value is **cq:Page**:

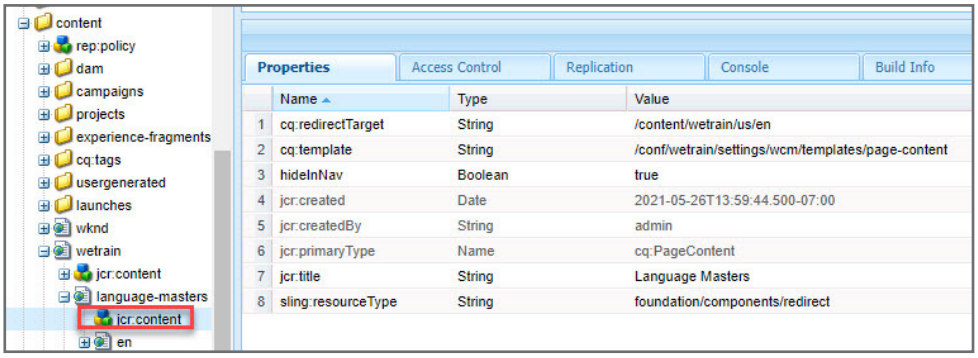


5. Select the **jcr:content** child node to see its properties, as shown:



   a. **cq:allowedTemplates** property dictates the templates that can be used under this location to create pages.
   b. **cq:conf** establishes the location to pull Cloud Configurations from and is inherited by sub-pages unless overridden.
   c. The various redirect properties are added as we do not envision this being a publicly accessible page.

6. Navigate to **/content/we-train/language-masters/jcr:content** to view its properties. At this stage, these are page properties defined in the template. In the future, these properties will be a mixture of those defined in the template and those defined for this page.



   a. Notice the value of **cq:template** is the "../page-content" Template provided by the archetype.
   b. Notice the value of **sling:resourceType**. Similar to the **sling:resourceType** value that you noticed in the earlier exercise for Structure, initial nodes of our template type and the resulting templates, it initiates rendering of the page.

Task 2: Create a Site Structure

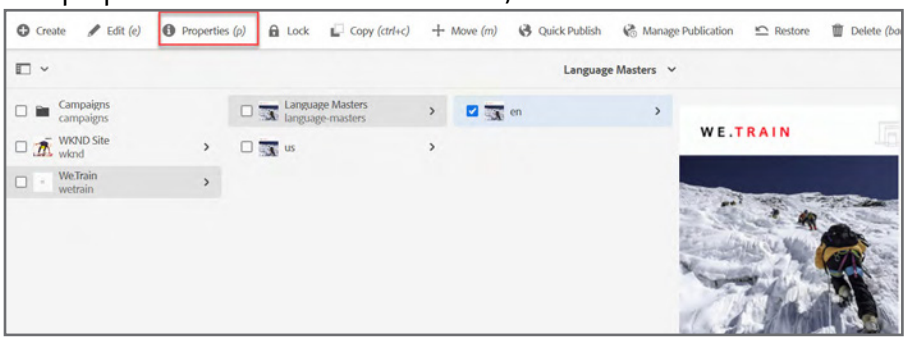
The content that the archetype built into the ../language-masters/en page is not something we need, so you will delete the pages we don't need and create a new site structure using the new WeTrain page template.

In this task, you will create the following website structure:

--> We.Train
-----> Language Masters
--------> English
----------> About Us
----------> Experiences
----------> Equipment
----------> Activities
----------> Hiking
----------> Biking
----------> Running
----------> Skiing
----------> Climbing
--------> French

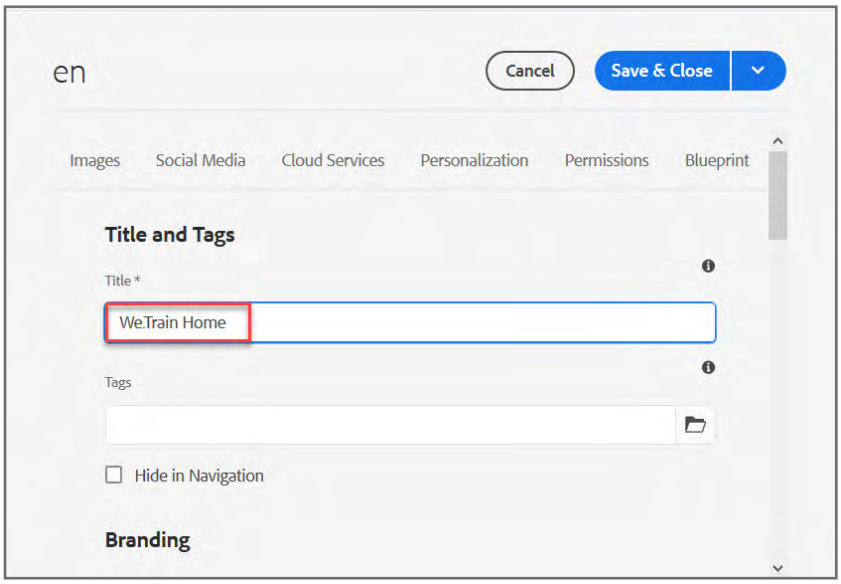
Note: The French page should already exist. This page was created earlier in the Custom Components module.

1. Click http://localhost:4502/sites.html/content. The **Sites** console page opens.
2. Navigate to **We.Train** > **Language Masters** and select the **en** page.
3. Click properties icon from the Actions bar, as shown:



The **en** properties page opens.

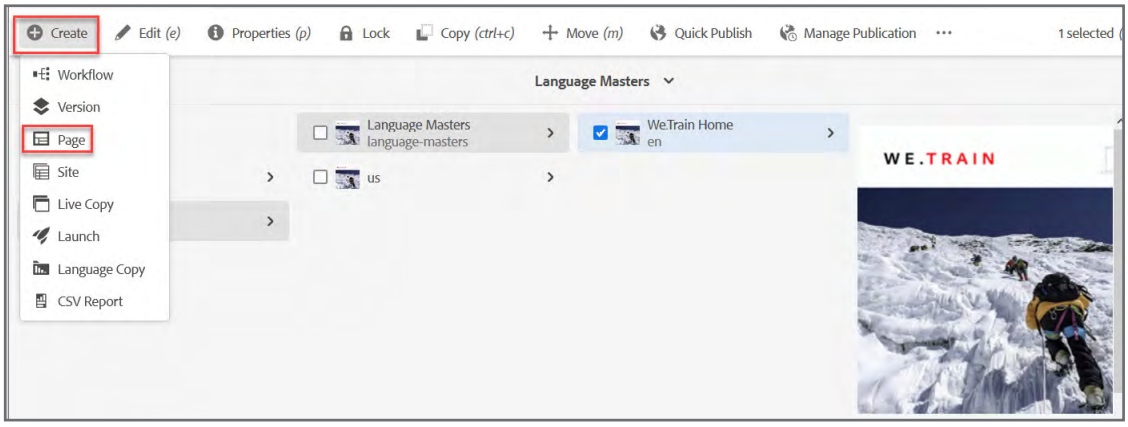
4. Assign the Title, as shown:
   › Title: **We.Train Home**



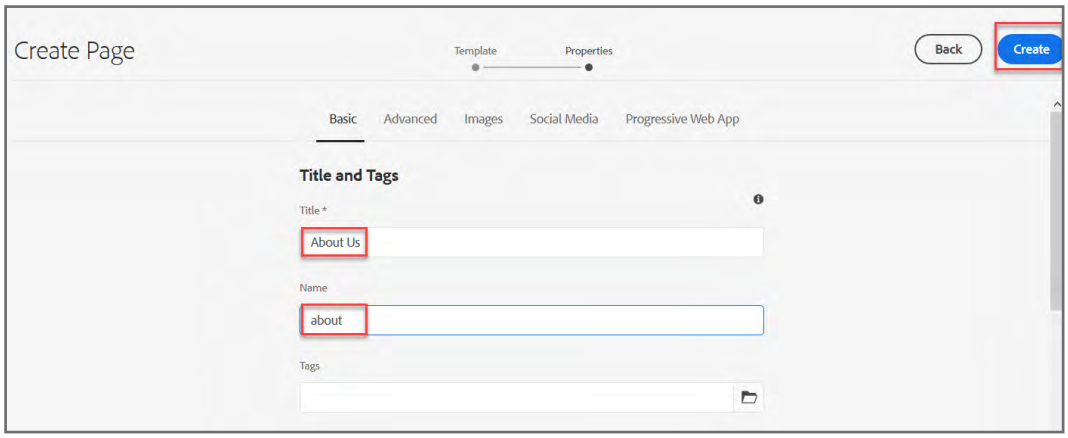
5. Click **Save & Close**.
   To create a page:

6. Navigate to **We.Train** > **Language Masters** > **We.Train Home.**

7. Select the **We.Train Home** checkbox and click **Create** > **Page** from the actions bar, as shown.
   The **Create Page** opens.



8. Select the **WeTrain Page** template and click **Next**. The **Properties** page opens.
9. Enter **About Us** in the **Title** box and **about** in the **Name** box, and then click **Create,** as shown. The **Success** dialog box opens.



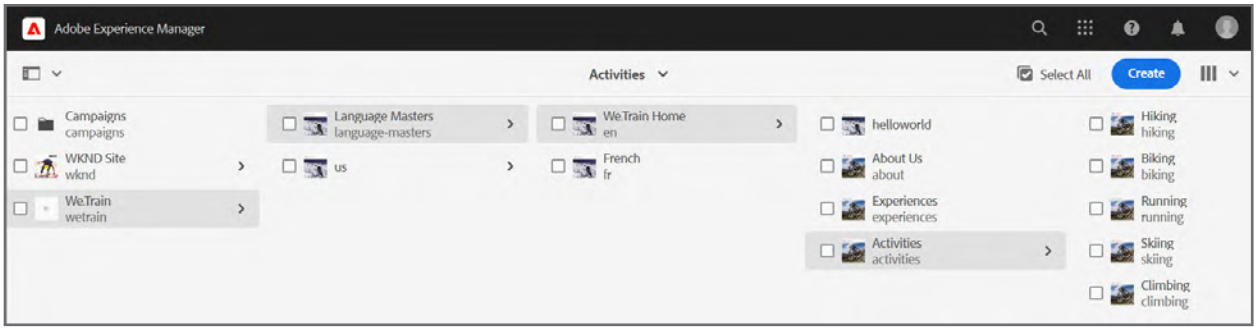
10. Click **Done**. The page is created.
11. Perform steps 6 through 10 and create the following website structure with the following naming conventions (where, T stands for Title and N for Name):

Note: If the **Name** is spelled the same as the **Title**, you can leave the **Name** field blank and AEM will auto-create it as a lowercased value. You do not need to type the **Name** separately.

--------> We.Train Home (T) (N: en)
----------> About Us (T) (N:)
----------> Experiences (T) (N: experiences)
----------> Equipment (T) (N: equipment)
----------> Activities (T) (N: activities)
----------> Hiking (T) (N: hiking)
----------> Biking (T) (N: biking)
----------> Running (T) (N: running)
----------> Skiing (T) (N: skiing)
----------> Climbing (T) (N: climbing)
--------> French (T) (N: fr)

Best Practice: Name all your pages and subpages in lowercase and use hyphens for two or more words.

After your site structure is complete, it should look similar to the site shown in the following screenshot:



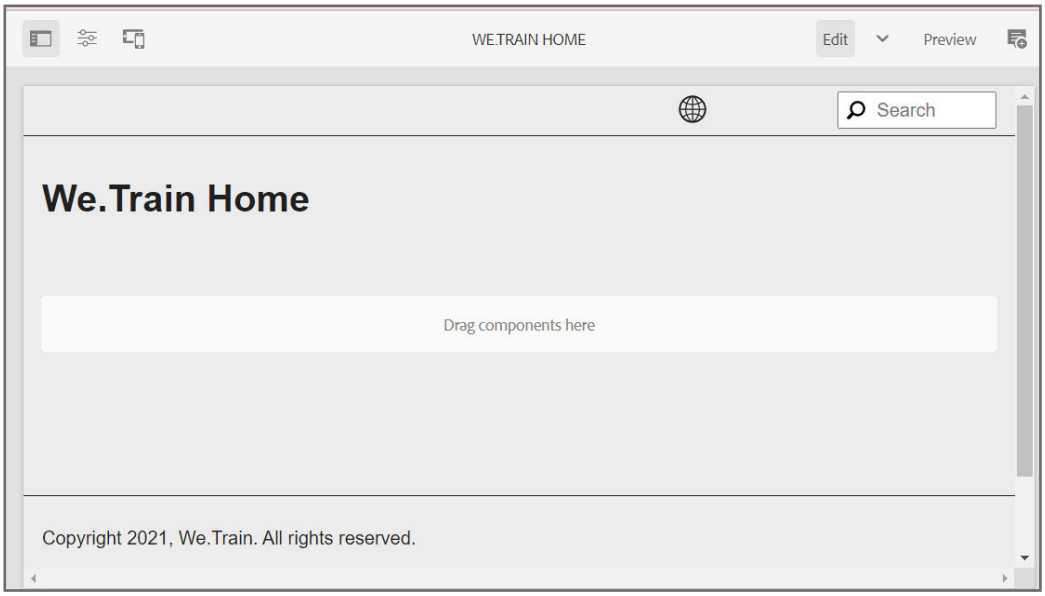
12. (Optional) You can upload a thumbnail to the content root. Notice that the content root page (/content/we-train) does not have a thumbnail in the Sites console. Add a thumbnail to the page by selecting the **We.Train** page, clicking **Properties**, and uploading an image from the **Thumbnail** tab.

Exercise 4: Author the Header and Footer

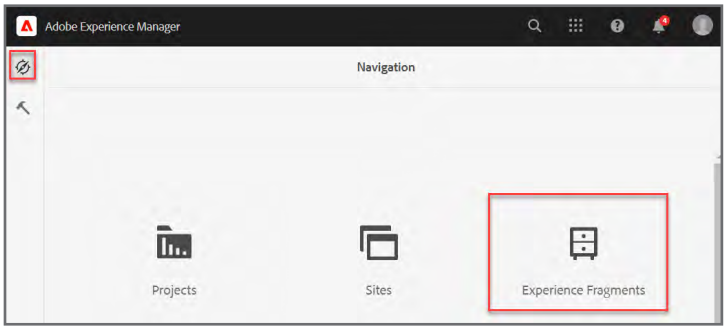
In a previous exercise, you added Experience Fragments to the template for the header and footer. In this exercise, you will learn how to make edits to these Experience Fragments and have the changes reflect on all pages using the header and footer fragments.

Task 1: Configure Content for Header/Footer

1. Go to http://localhost:4502/sites.html/content. The **Sites** console page opens.
2. Navigate to **We.Train** > **Language Masters** and select **We.Train Home** page.
3. Click the Edit icon from the Actions bar. The **We.Train Home** page opens in edit mode. Notice the Header and Footer output for this page.



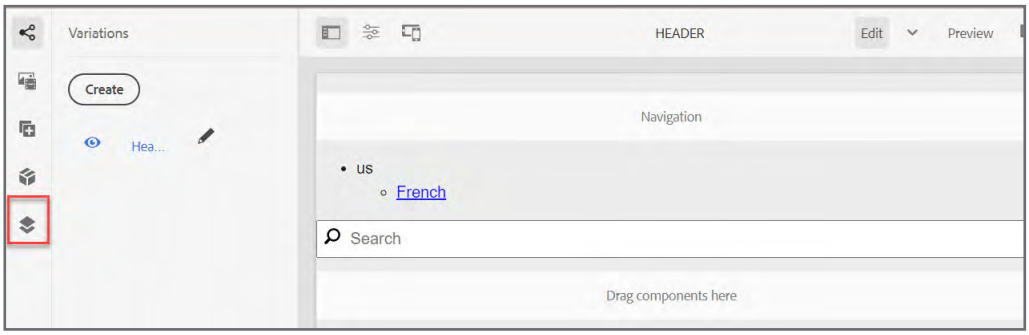
4. Navigate to **Adobe Experience Manager** > **Experience Fragments**, as shown:



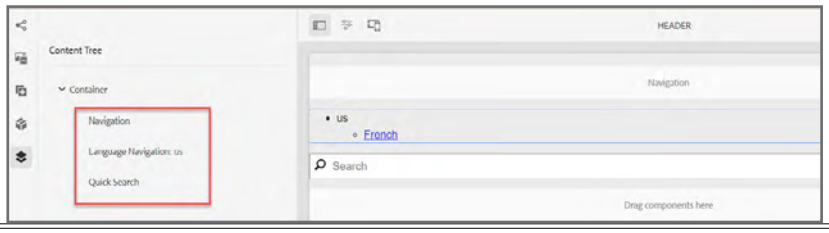
5. Navigate to **We.Train Fragments** > **Language Masters** > **en** > **Site** > **Header**. select **Header** and click Edit, as shown:

Note: The header and footer experience fragments were generated from the archetype our Maven project came from.

6. The side panel should open by default. Expand it if not. Select the content tree, as shown:

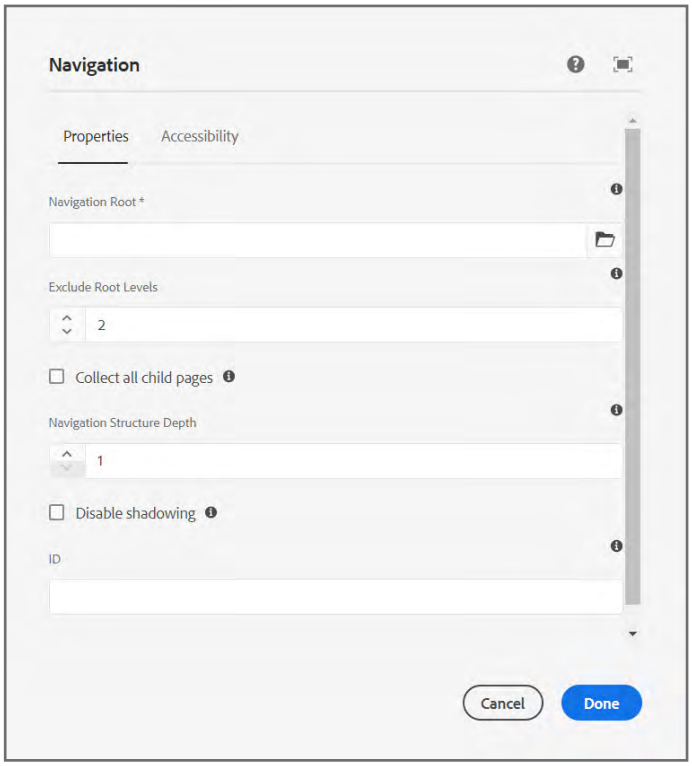


7. Notice the components the archetype has built in: **Navigation**, **Language Navigation**, and **Quick Search**, as shown:

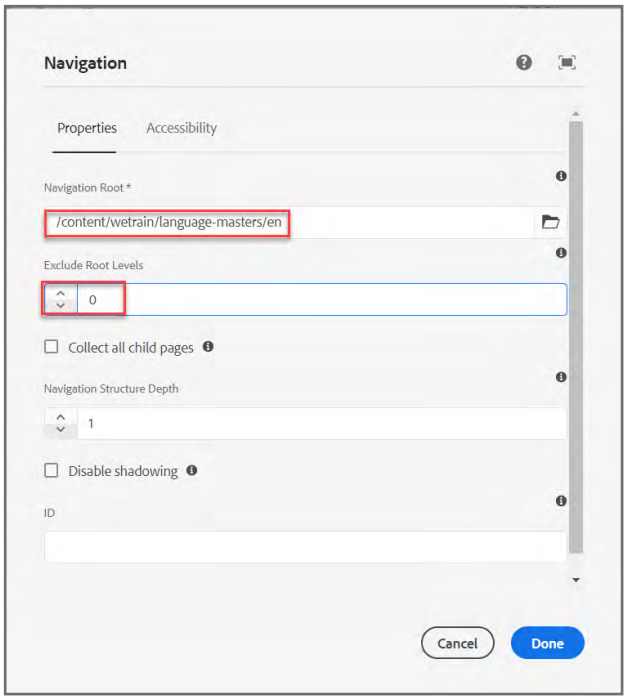


Note: Navigation was not showing at all on the **We.Train** > **Language Masters** > **en** page because it is not yet configured.

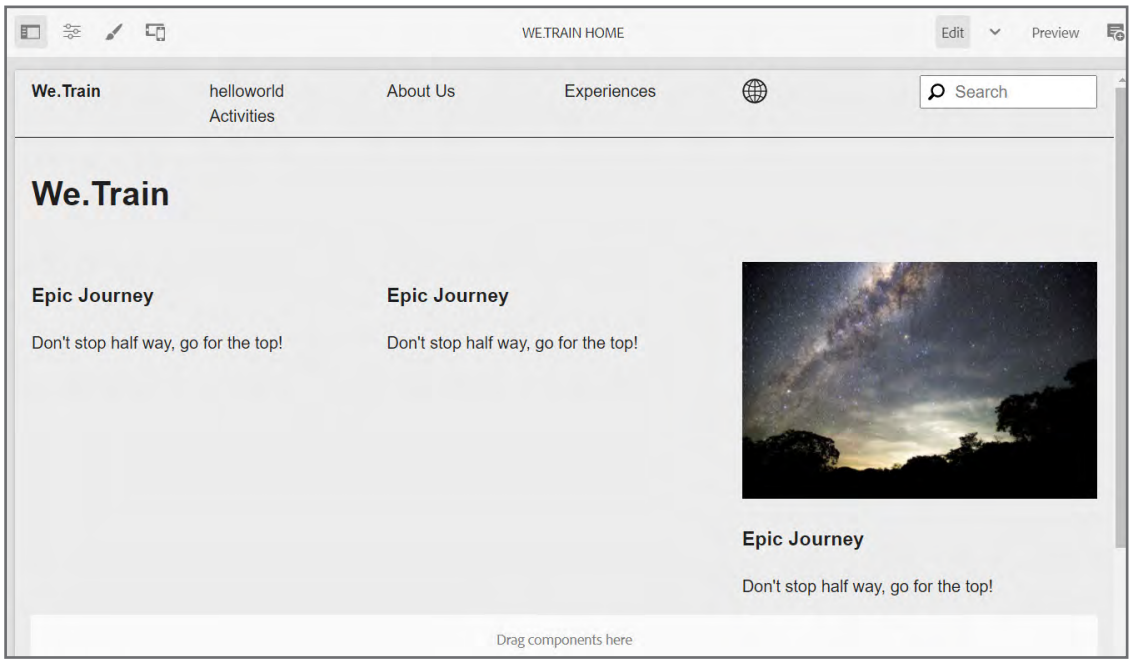
8. Hover over the **Navigation** component and select the Wrench icon to configure. The **Navigation** dialog opens, as shown:



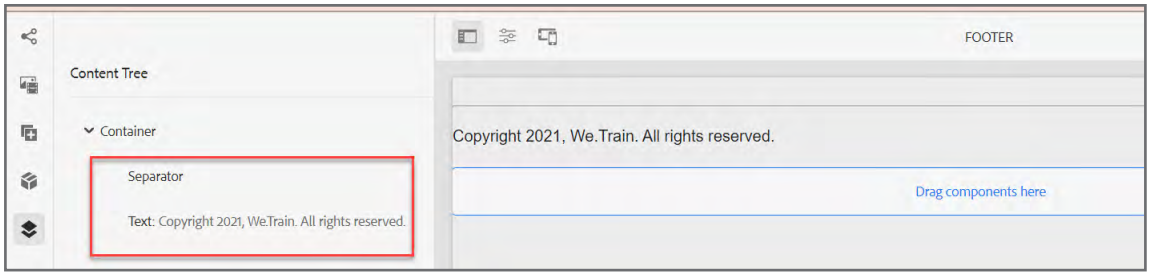
9. Configure Navigation, as shown:
   - Navigation Root: **/content/wetrain/language-masters/en**
   - Exclude Root Levels: **0**



10. Click **Done** to save the changes.
11. Hover over the **Language Navigation** component and select the Wrench icon to configure. The **Language Navigation** dialog opens.
12. Set **Navigation Root** to **/content/wetrain/language-masters**.
13. Click **Done** to save the changes.
14. Go back to the browser tab you opened to **We.Train** > **Language Masters** > **We.Train Home** page and refresh the browser.

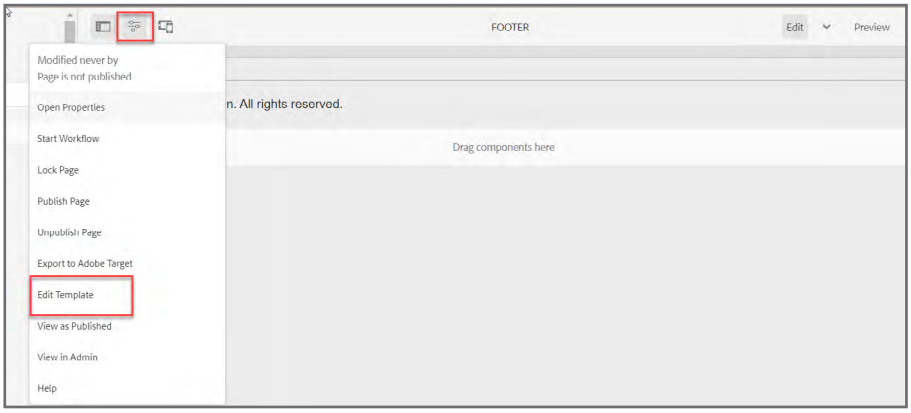


15. Notice how the navigation is now dynamically appearing in the header.
16. In your browser, open the AEM homepage and navigate to **Experience Fragments** > **We.Train Fragments** > **Language Masters** > **en** > **Site** > **Footer.**
17. Select **Footer** and click **Edit**.
18. The side panel open by default. Expand it if not already opened. Select the **Content Tree**.
19. Notice the components that the archetype has built: **Separator** and **Text**, as shown:



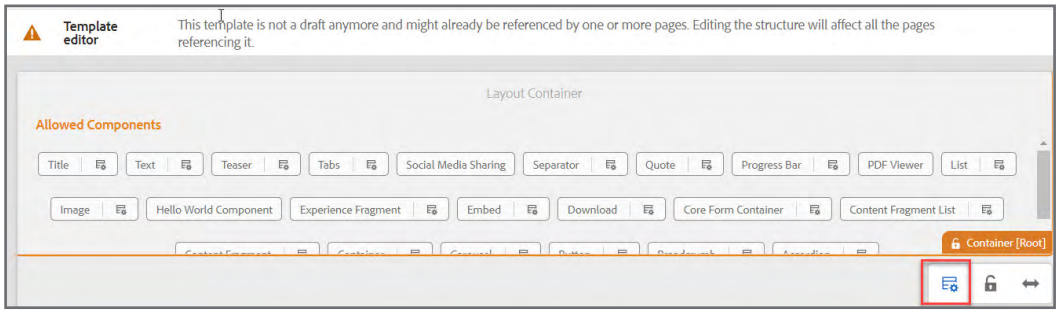
20. We want to add Navigation here as well, but look at the list of available **Components** in the Side Panel. Navigation is not one of them. We will need to first add it to the Template.

21. Select the Page Information icon at the top and click **Edit Template**, as shown:



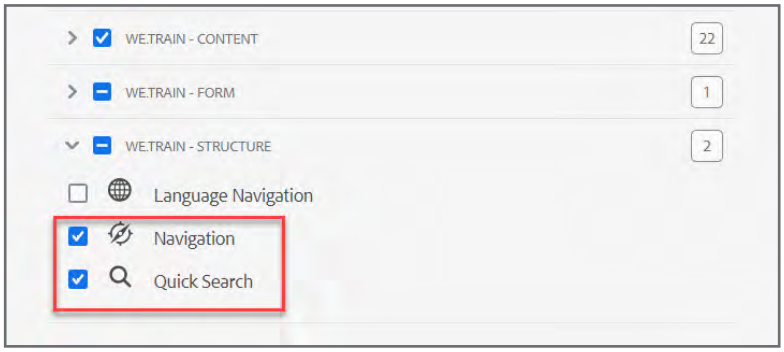
The **Template editor** opens.

22. Select the **Container** with **Allowed Components** and click its **Policy** icon, as shown:



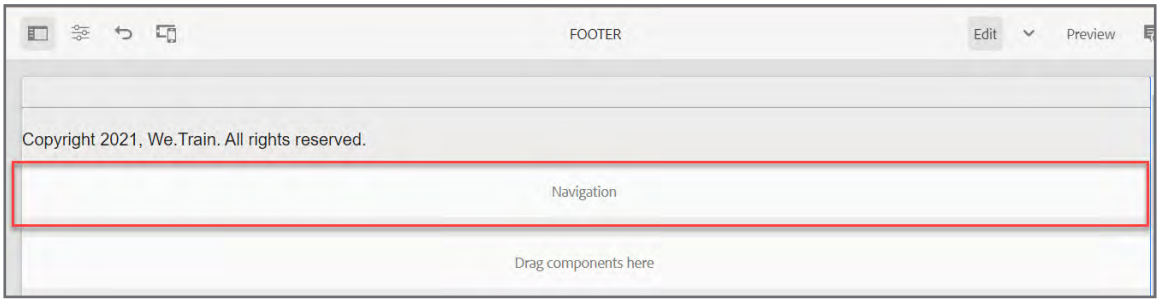
23. On the right side, scroll down the list of **Allowed Components** to find the **We.Train – Structure** group.

24. Expand the group and select **Navigation** and **Quick Search**, as shown:



Note: We are not adding Search to the Footer for now, but we will use it in other Experience Fragments in this project.

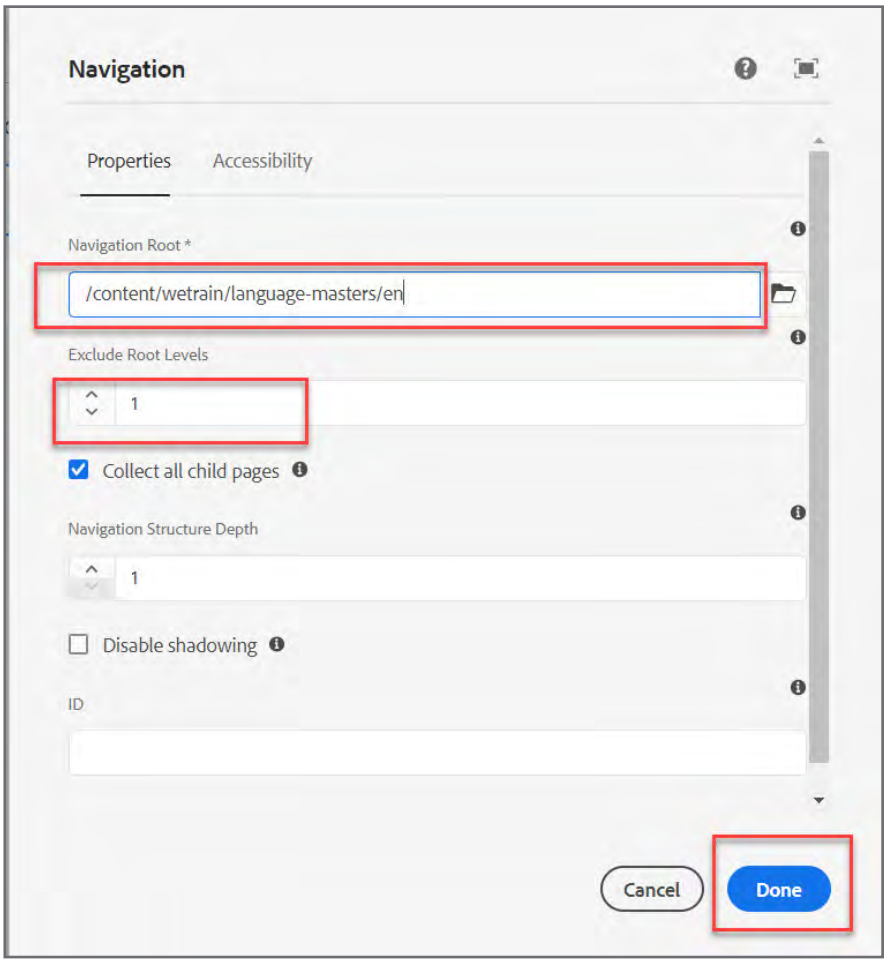
25. Click **Done** to save the Template editor.
26. Go back to the browser tab with the Footer experience fragment and refresh the browser.
27. Go to the **Components** tab in the side panel. Drag and drop the **Navigation** component below the text component, as shown:



28. Hover over the **Navigation** component and select the Wrench icon to configure. The **Navigation** dialog opens.

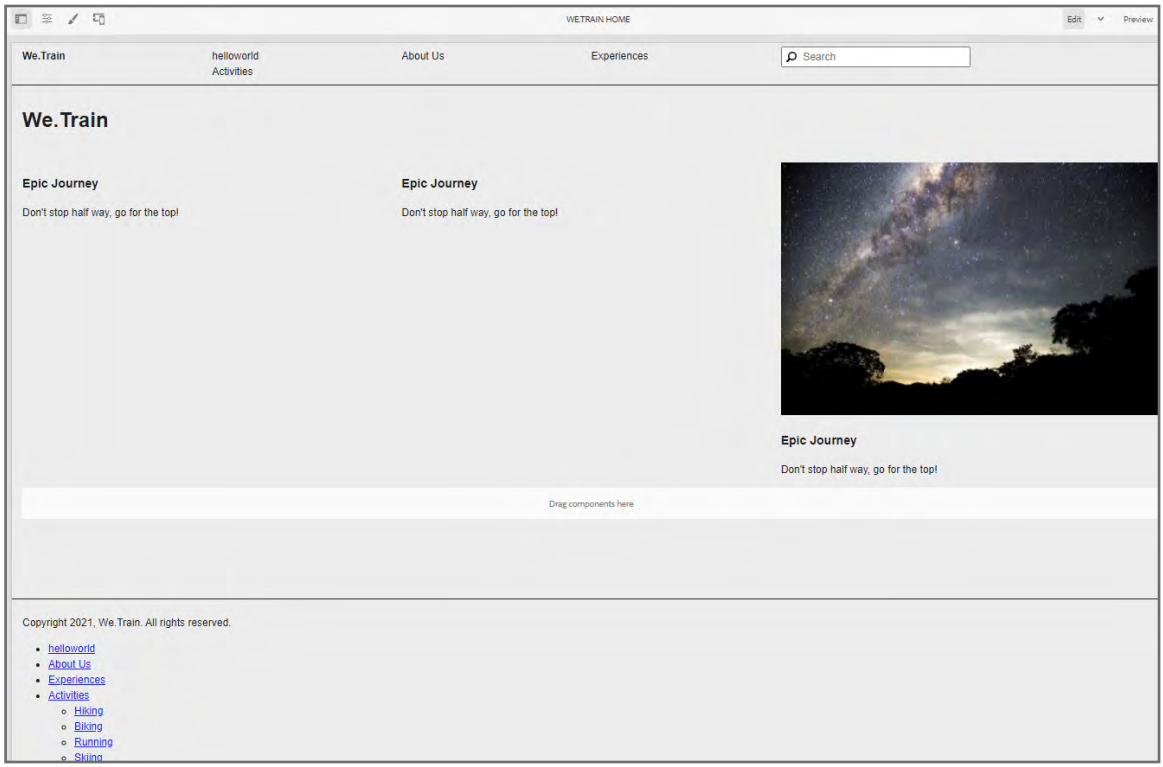
29. Configure the Navigation **Properties**:
    - Navigation Root: **/content/wetrain/language-masters/en** (click open selection dialog icon and choose)
    - Exclude Root Levels: 1
    - Click **Done**

Note: We will not have Navigation display the We.Train (home) page as we did in the Header, so we will set the "Exclude Root Levels" field to 1.



30. Go to the browser tab that is open to the **We.Train Home** website page and refresh it. (http://localhost:4502/editor.html/content/wetrain/language-masters/en.html)

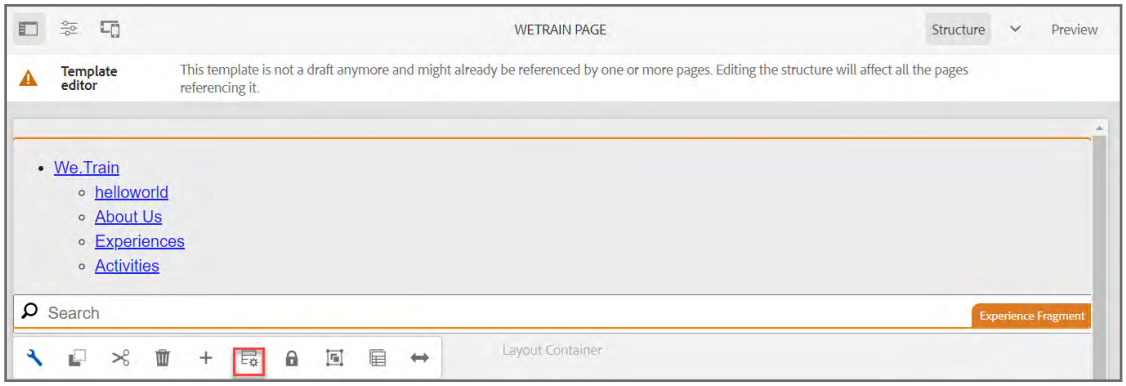
Your page should look similar to the screenshot below:



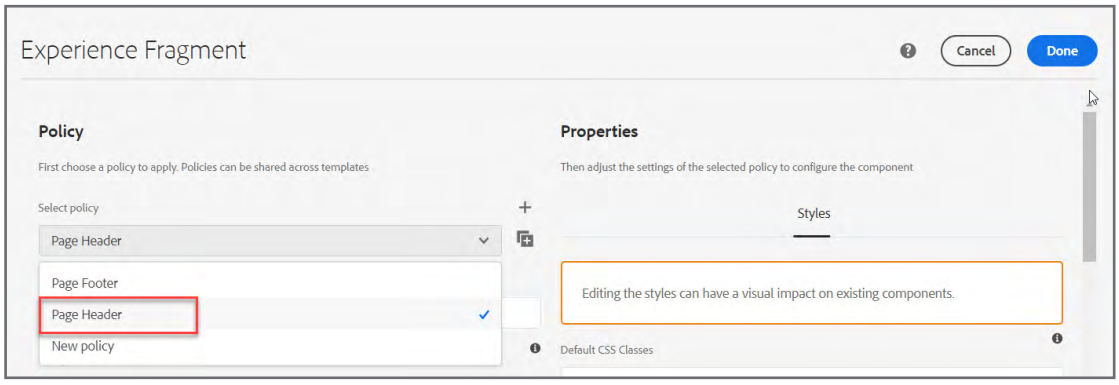
We have content in the header and footer.

Task 2: Style the Header/Footer

1. If you are not already in the Template page, navigate to the **Adobe Experience Manager** > **Tools** > **Templates** > **We.Train** folder.
2. Click the **We.Train** folder to open it. The Template page opens.
3. Select the **We.Train** Page template and click **Edit** from the actions bar. The Template editor opens.
4. Select the **Experience Fragment** at the top of the page and click the policy icon, as shown:



5. In the Policy section, select the **Page Header** policy, as shown, from the Select policy dropdown menu.



Note: Notice this sets the default element to <header>. This will connect this **Experience Fragment** with rules defined in our existing experiencefragment_header.scss file.

6. Click **Done**. The policy is updated.
7. Select the **Experience Fragment** at the bottom of the page and click the policy icon.
8. In the Policy section, select the **Page Footer** policy from the Select policy dropdown menu.

Note: Notice this sets the default element to <footer>. This will connect this **Experience Fragment** with rules defined in our existing experiencefragment_footer.scss file.

9. Click **Done**. The policy is updated.
10. In your IDE, navigate to **ui.frontend/src/main/webpack/site/styles.**
11. Open the **experiencefragment_header.scss** file.
12. In the Exercise_Files-DWC folder, navigate to **ui.frontend/src/main/webpack/site/styles/**.
13. Copy contents of the **experiencefragment_header.scss** file and paste it into the **experiencefragment_header.scss** file.
14. Similarly, copy the contents of the **experiencefragment_footer.scss** file and paste it into the **experiencefragment_footer.scss** file.
15. Save the changes (**File** > **Save**).

Task 3: Deploy and Verify

1. In your IDE, open a terminal window.
2. Type in the following command to deploy your project to AEM:

```
mvn clean install -PautoInstallSinglePackage
```

3. Observe changes on an AEM page in our project.

Exercise 5: Create a Template Type with Header and Footer

After testing all structure components (assuming they have gone through a full QA process), you can complete the template type.
In this exercise, you will enable template authors to use the content components in the template along with the structure components.
You will finish this project by finalizing two page template types for the project:

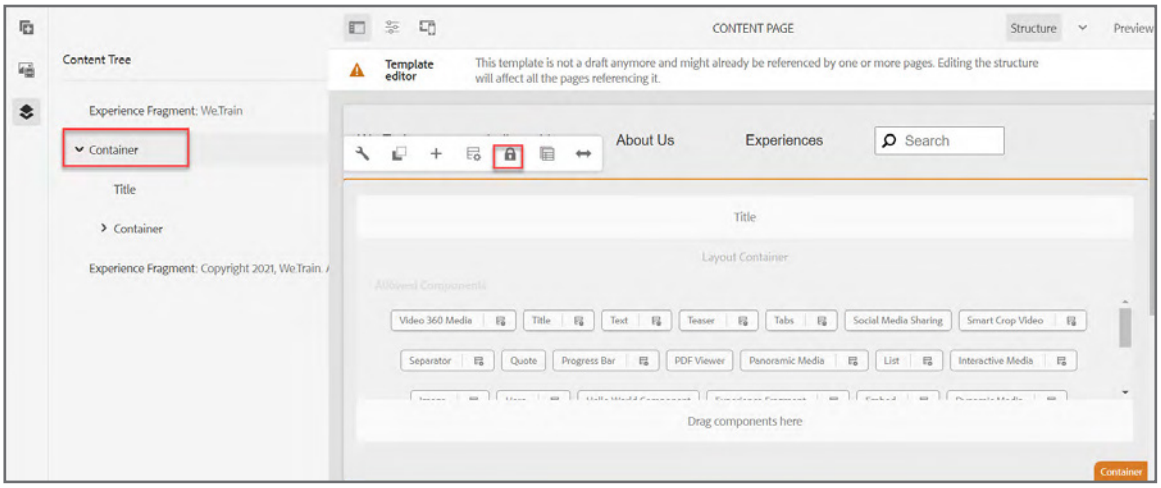
- The original template type which has no structure other than the initial container it provides.
- The template type that starts a template with a header and footer already configured.

Task 1: Add Template Settings to Original Template Type

The work you have done in either template (addition of Experience Fragments, setting of policies, etc…) is unique to the given Template. Each time a new template is created, those steps must be repeated. To avoid this repetition, you can move those settings to the template type.

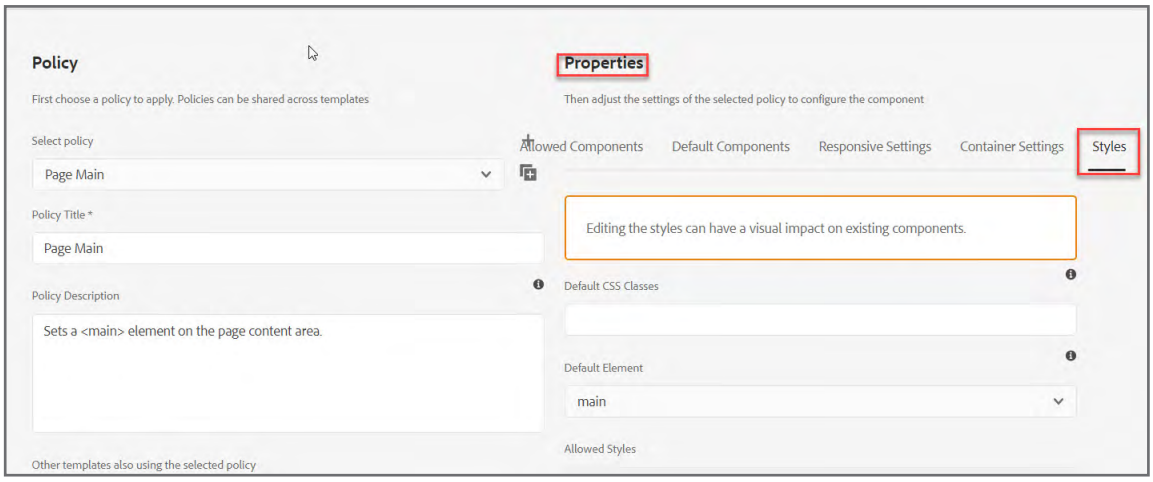
1. If you are not already in the Template page, navigate to **Adobe Experience Manager** > **Tools** > **Templates** > **We.Train** folder.
2. Click the **We.Train** folder to open it. The Template page opens.
3. Select the **Content Page** template and click **Edit** from the actions bar. The Template editor opens.
4. Click the Toggle Side Panel icon from the toolbar. The panel opens.
5. Click the Content Tree icon. The **Content Tree** panel opens.
6. Expand the container to verify it contains a **Title** and a **Container** component. This inner **Container** component is the one configured for the authors to use on a page.

7. Click the top level **Container** in the **Content Tree** to expose its border and component toolbar. Notice the **Container** is locked.



8. Click the Policy icon. The Container wizard opens.

9. Select the **Styles** tab under the **Properties** section. Notice this policy sets a **main** element on the page content area.



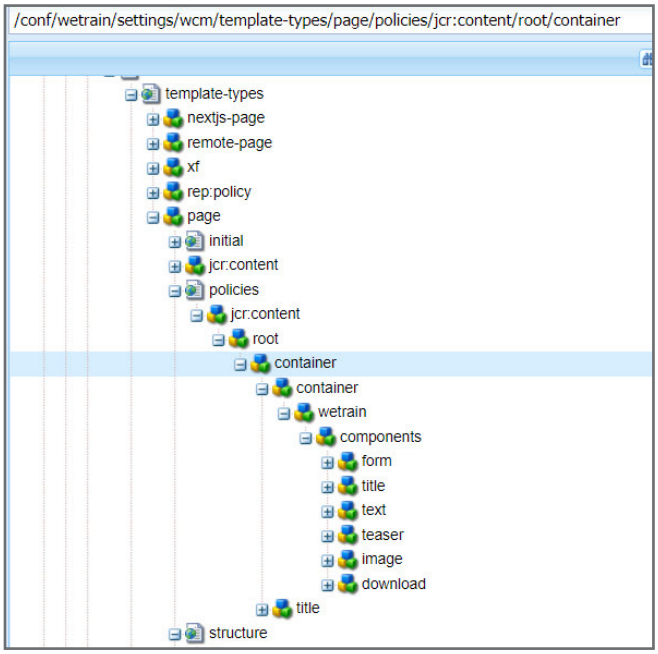
10. In your IDE, navigate to **ui.frontend/src/main/webpack/site/styles** and examine the file **container_main.scss**.

```
ui.frontend > src > main > webpack > site > styles > container_main.scss > ...
1
2   //== Container main content style, used on page template
3
4   main.container {
5       padding: .5em 1em;
6   }
7
```

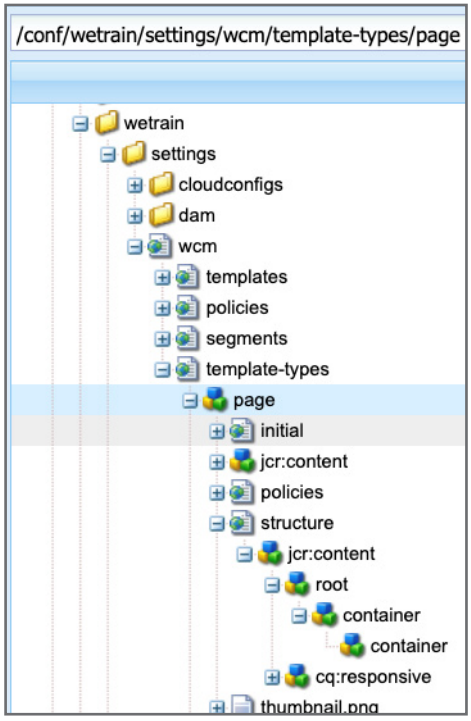
The above simple rule adds padding to all page content added by the Authors.

Note: The archetype builds this rule into the content page template. It is not in the template type. Adding it to the template type ensures that the newly created templates have the same setting.

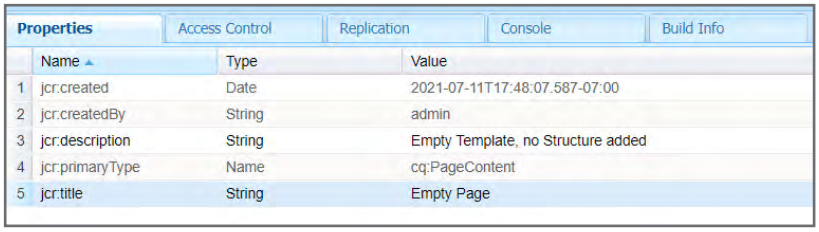
11. In **CRXDE Lite**, navigate to **/conf/wetrain/settings/wcm/templates/page-content/policies/jcr:content/root/container**.
12. Right-click **container** and select **Copy**.
13. Navigate to **/conf/wetrain/settings/wcm/template-types/page/policies/jcr:content/root** and delete the **container** node.
14. Click **Save All**.
15. Right-click the **root** node and paste in the copied **container** node.
16. Expand the container node and its children. You should see the following structure:



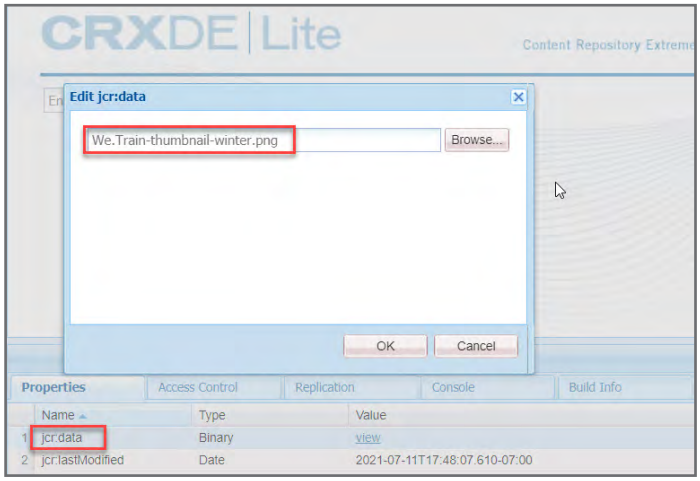
17. Delete the **title** node which is a sibling of the second-level container node as you are not adding a Structure mode Title component to the template type.
18. Save the changes.
19. In **CRXDE Lite**, navigate to **/conf/wetrain/settings/wcm/templates/page-content/structure/jcr:content/root/container**. Copy the node **container**.
20. Navigate to **/conf/wetrain/settings/wcm/template-types/page/structure/jcr:content/root**.
21. Delete the **container** node under **root**.
22. Click **Save All**.
23. Right-click the **root** node and paste in the copied **container** node.
24. Expand the **container** node and delete the title node. Save All.
25. The node structure should look as shown:



26. Select the **conf/wetrain/settings/wcm/template-types/page/jcr:content** node.
27. In the **Properties** tab, update the value of **jcr:title** to **Empty Page.**
28. Click **Save All**.
29. Update the value of **jcr:description** to **Empty Template, no Structure added** and save the changes.



30. Expand **thumbnail.png** node and select **jcr:content** node.
31. In the Properties tab, double-click **jcr:data** in the Name column. The **Edit:jcr:data** dialog opens.
32. Browse and upload **../training-files/editable-templates/We.Train-thumbnail-winter.png** from the Exercise_Files-DWC folder.

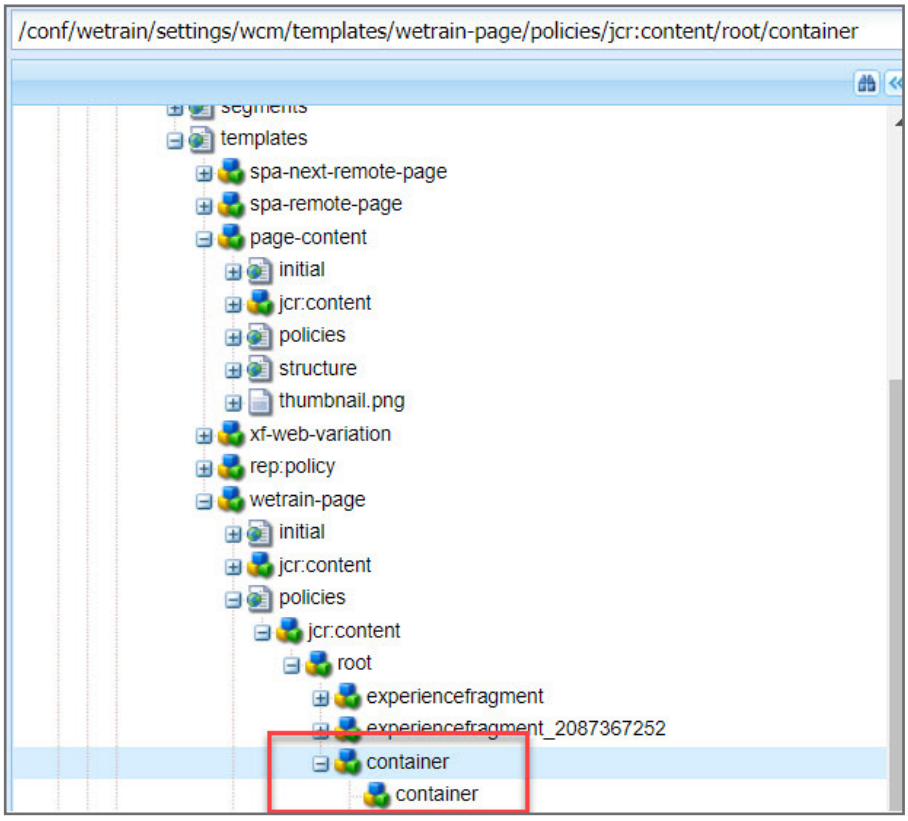


33. Click **OK**.
34. Test the changes by creating a new template from the page template type. It should have a container for page authors with components available and an outer container with the **Page Main** policy set on it.

Task 2: (Optional) Create a New Template Type

Note: With the skills you learned in the last Task, you can now build out the structure and policies of any Template Type that you need. This optional task will walk you through the process.

1. In CRXDE Lite, navigate to **/conf/wetrain/settings/wcm/templates/wetrain-page/policies/jcr:content/root/container**.
2. Notice it does not have a child container node like the **page-content** template you were working with in the previous task. This is because you created the **wetrain-page** template before saving the settings back into the page template type.
   To fix, you will copy the node structure from the page-content template:
3. Navigate to **/conf/wetrain/settings/wcm/templates/page-content/policies/jcr:content/root/container.** Copy the node **container.**
4. Navigate to **/conf/wetrain/settings/wcm/templates/wetrain-page/policies/jcr:content/root**. Delete the **container** child node.
5. Right-click the **root** node and paste in the copied **container** node from Step 4.
6. Expand the **container** node child structure until you see the **title** node. Delete the **title** node.

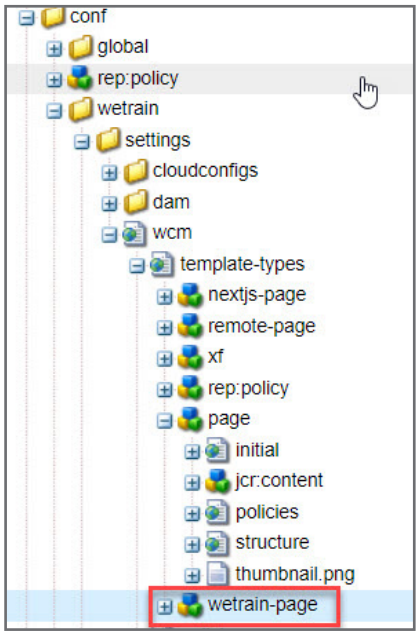


7. Navigate to **/conf/wetrain/settings/wcm/templates/page-content/structure/jcr:content/root/container.** Copy the **container** node.
8. Navigate to **/conf/wetrain/settings/wcm/templates/wetrain-page/structure/jcr:content/root.** Delete the **container** child node.
9. Click **Save All.**
10. Right-click the **root** node and paste in the copied **container** node from Step 7.
11. Expand the **container** node child structure. Delete the **title** node.

Note: We are not including a Title component as part of the structure for the WeTrain page template, so we deleted both the structure and the policy setting for this Title component.

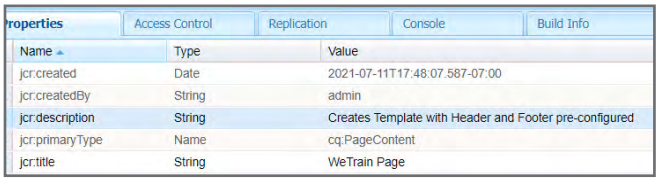
Next you will create your new template type from the page template type that you worked on in the previous task. To do this:

12. Copy the **/conf/wetrain/settings/wcm/template-types/page** node.
13. Right-click the parent **template-types** node and paste.
14. Rename the newly pasted node name to **wetrain-page**.



15. Select the **wetrain-page/jcr:content** node. Update the value of **jcr:title** property to **WeTrain Page.**

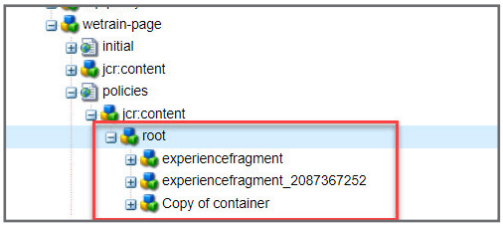
16. Update the value of **jcr:description** to **Creates Template with Header and Footer pre-configured**.



17. Click **Save All**.

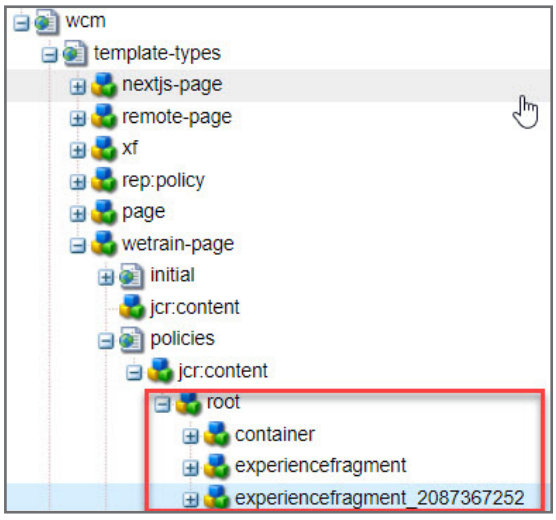
Now you need to bring the policies and structure which you built into the WeTrain Page template into the template type. To do this:

18. Navigate to **/conf/wetrain/settings/wcm/templates/wetrain-page/policies/jcr:content/root.** Notice two **experiencefragment** nodes:



19. Copy the first **experiencefragment** node and paste it under: **/conf/wetrain/settings/wcm/template-types/wetrain-page/policies/jcr:content/root.**

20. Similarly, copy the second **experiencefragment_<nnnnnn>** node and paste it under: **/conf/wetrain/settings/wcm/template-types/wetrain-page/policies/jcr:content/root.**

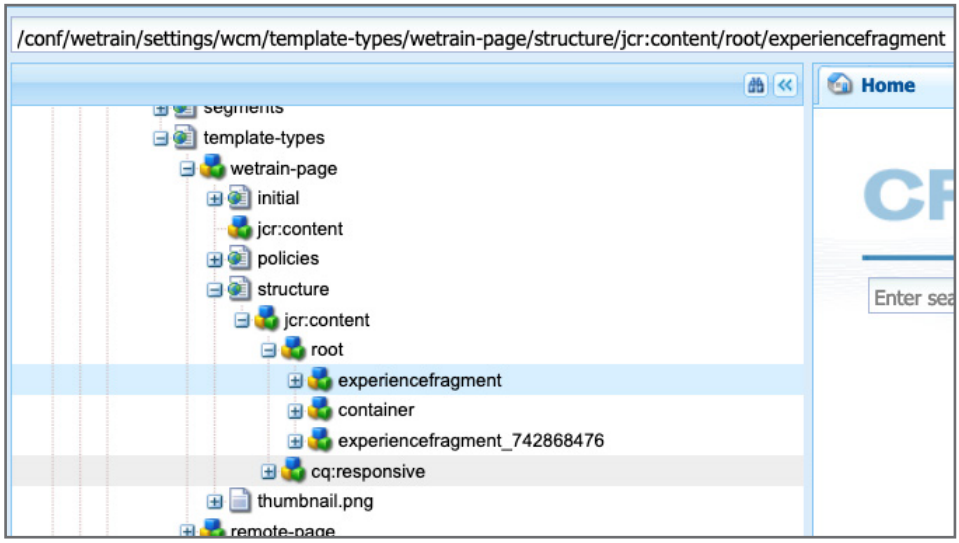


21. Similarly, perform the following actions:
    a. Navigate to **/conf/wetrain/settings/wcm/templates/wetrain-page/structure/jcr:content/root.**
    b. Copy the two **experiencefragment** nodes.
    c. Paste them under: **/conf/wetrain/settings/wcm/template-types/wetrain-page/structure/jcr:content/root**.

Note: You can copy only one **experiencefragment** node at a time.

Unlike the policies node, the order of the nodes under structure matters. To fix the order:

22. Click and hold the first experiencefragment node (the one without the extra numeric ID). Drag it just above the container node but still below the parent **root** node, as shown:



23. Test by creating a new template from this template type.
    It should have all structure and policies already set.