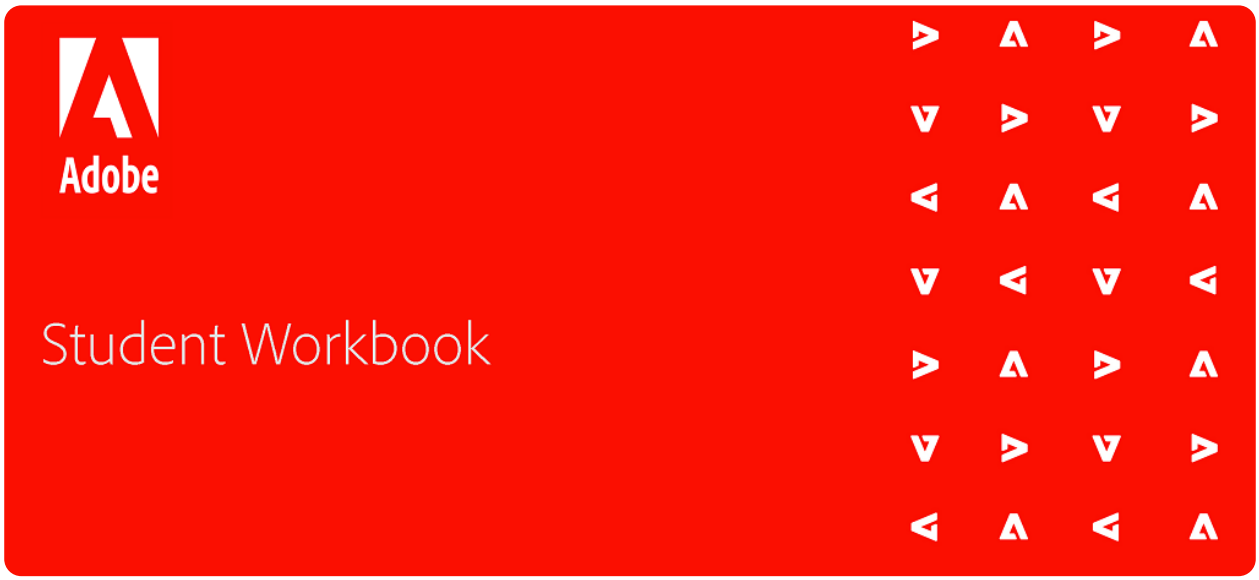
Develop Single-Page Applications with React in AEM



Find your journey at learning.adobe.com >

Develop Single-Page Applications with React in AEM

Adobe, 345 Park Avenue, San Jose, California 95110, USA.

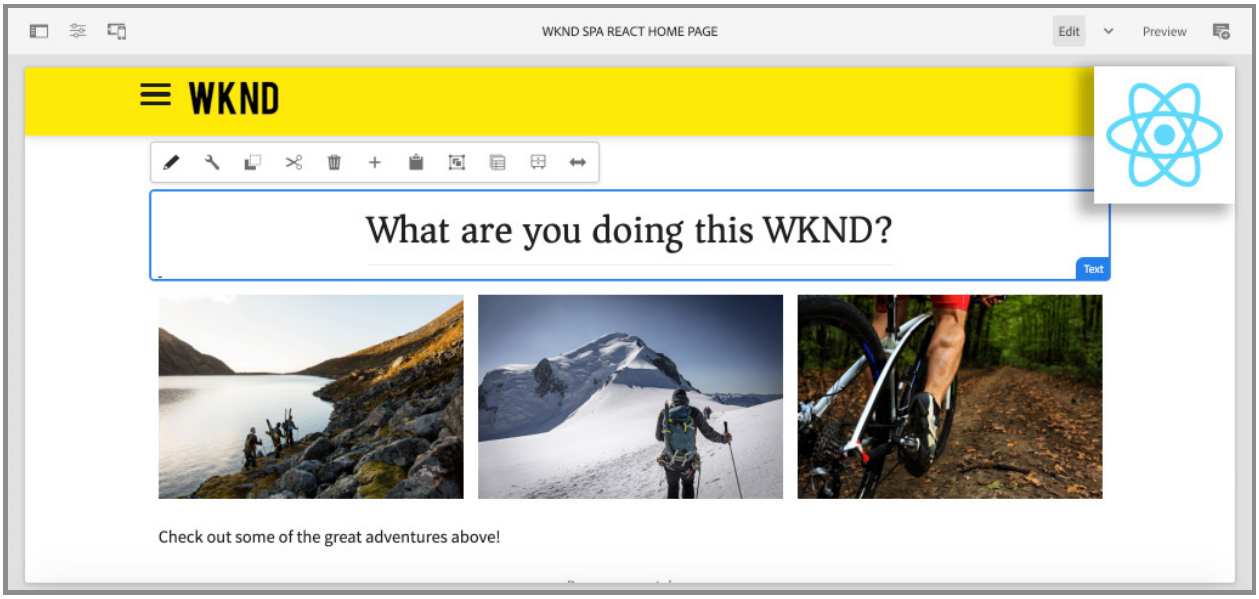
12/10/2021

Course Contents

Extend a Core Component

Create Your First React SPA in AEM

Welcome to a multi-part course designed for developers new to the **SPA Editor** feature in Adobe Experience Manager (AEM). This course walks through the implementation of a React application for a fictitious lifestyle brand, the WKND. The React app will be developed and designed to be deployed with AEM's SPA Editor, which maps React components to AEM components. The completed SPA, deployed to AEM, can be dynamically authored with traditional in-line editing tools of AEM.



WKND SPA Implementation

About

The course is designed to work with **AEM as a Cloud Service** and is backwards compatible with **AEM 6.5.4+** and **AEM 6.4.8+**.

Latest Code

The solution files for each module in this student guide are available in the accompanying **Exercise Files-spa** > **Solutions** folder. In the same **Exercise Files-spa** folder, you are also given a starter and completed project. All of this code can also be found in the public github repository: https://github.com/adobe/aem-guides-wknd-spa.

Prerequisites

Before starting this course, you'll need the following:

- A basic knowledge of HTML, CSS, and JavaScript
- Basic familiarity with React

While not required, it is beneficial to have a basic understanding of developing traditional AEM Sites components.

Local Development Environment

A local development environment is necessary to complete this course. Screenshots and video are captured using the AEM as a Cloud Service SDK running on a Mac OS environment with Visual Studio Code as the IDE. Commands and code should be independent of the local operating system, unless otherwise noted.

Required Software

- AEM as a Cloud Service SDK, AEM 6.5.4+ or AEM 6.4.8+
- Java
- Apache Maven (3.3.9 or newer)
- Node.js and npm

NOTE:

New to AEM as a Cloud Service? Check out the following guide to setting up a local development environment using the AEM as a Cloud Service SDK.

New to AEM 6.5? Check out the following guide to setting up a local development environment.

Create Project

Learn how to generate an Adobe Experience Manager (AEM) Maven project as a starting point for a React application integrated with the AEM SPA Editor.

Objectives

1. Generate a SPA Editor enabled project using the AEM Project Archetype.
2. Deploy the starter project to a local instance of AEM.

What You Will Build

In this chapter, a new AEM project will be generated, based on the AEM Project Archetype. The AEM project will be bootstrapped with a very simple starting point for the React SPA.

What is a Maven project? - Apache Maven is a software management tool to build projects. *All Adobe Experience Manager* implementations use Maven projects to build, manage and deploy custom code on top of AEM.

What is a Maven archetype? - A Maven archetype is a template or pattern for generating new projects. The AEM Project archetype allows us to generate a new project with a custom namespace and include a project structure that follows best practices, greatly accelerating our project.

Prerequisites

Review the required tooling and instructions for setting up a local development environment. Ensure that a fresh instance of Adobe Experience Manager, started in **author** mode, is running locally.

Create the Project

NOTE:

This course uses version **27** of the archetype. It is always a best practice to use the **latest** version of the archetype to generate a new project.

1. Open up a command line terminal and enter the following Maven command:

```
mvn -B archetype:generate \
 -D archetypeGroupId=com.adobe.aem \
 -D archetypeArtifactId=aem-project-archetype \
 -D archetypeVersion=27 \
 -D appTitle="WKND SPA React" \
 -D appId="wknd-spa-react" \
 -D artifactId="aem-guides-wknd-spa.react" \
 -D groupId="com.adobe.aem.guides.wkndspa.react" \
 -D frontendModule="react" \
 -D aemVersion="cloud"
```

NOTE:

If targeting AEM 6.5.5+ replace `aemVersion="cloud"` with `aemVersion="6.5.5"`. If targeting 6.4.8+, use `aemVersion="6.4.8"`.

Notice the `frontendModule=react` property. This tells the AEM Project Archetype to bootstrap the project with a starter React code base to be used with the AEM SPA Editor. Properties like `appTitle`, `appId`, `artifactId`, and `groupId` are used to identify the project and purpose.

A full list of available properties for configuring a project can be found here.

2. The following folder and file structure will be generated by the Maven archetype on your local file system:

```
|--- aem-guides-wknd-spa.react/
    |--- all/
    |--- analyse/
    |--- core/
    |--- ui.apps/
    |--- ui.apps.structure/
    |--- ui.config/
    |--- ui.content/
    |--- ui.frontend/
    |--- ui.tests /
    |--- it.tests/
    |--- dispatcher/
    |--- analyse/
    |--- pom.xml
    |--- README.md
    |--- .gitignore
```

Each folder represents an individual Maven module. In this course we will primarily be working with the `ui.frontend` module, which is the React app. More details about individual modules can be found in the AEM Project Archetype documentation.

Deploy and Build the Project

Next, compile, build, and deploy the project code to a local instance of AEM using Maven.

1. Ensure an instance of AEM is running locally on port **4502**.
2. From the command line navigate into the `aem-guides-wknd-spa.react` project directory.

   ```
   $ cd aem-guides-wknd-spa.react
   ```

3. Run the following command to build and deploy the entire project to AEM:

   ```
   $ mvn clean install -PautoInstallSinglePackage
   ```

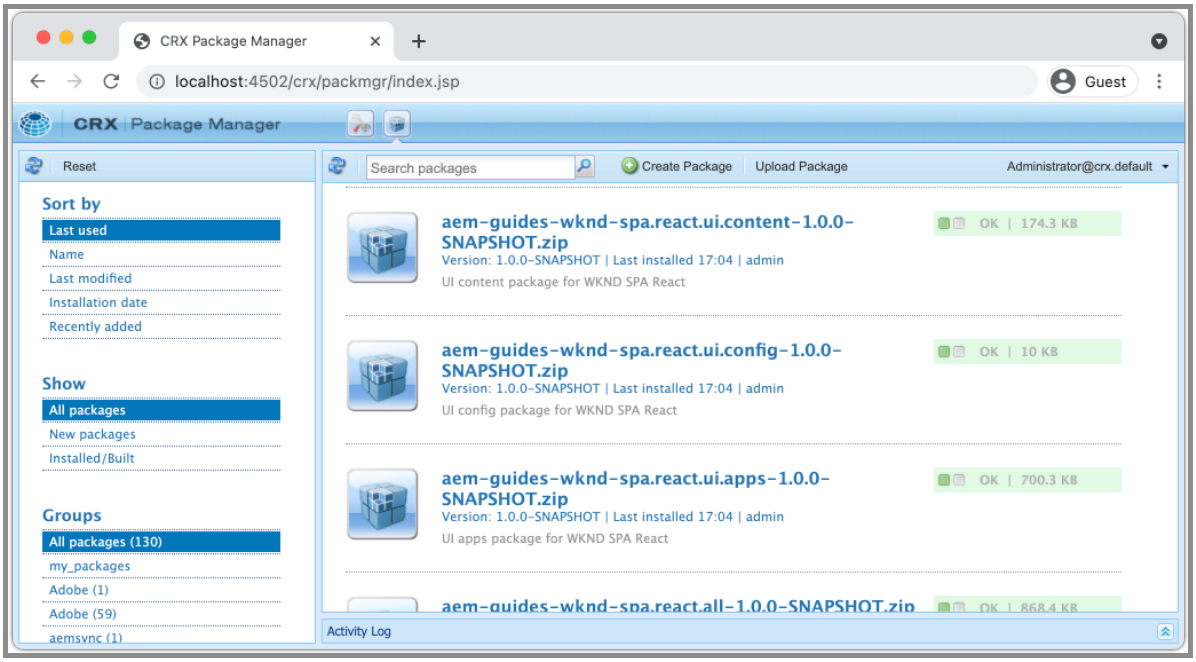
   The build will take around a minute and should end with the following message:

```
...
[INFO] ------------------------------------------------------------
-------------
[INFO] Reactor Summary for aem-guides-wknd-spa.react 1.0.0-
SNAPSHOT:
[INFO]
[INFO] aem-guides-wknd-spa.react ..........................
SUCCESS [  0.257 s]
[INFO] WKND SPA React - Core ..............................
SUCCESS [ 12.553 s]
[INFO] WKND SPA React - UI Frontend .......................
SUCCESS [01:46 min]
[INFO] WKND SPA React - Repository Structure Package ......
SUCCESS [  1.082 s]
[INFO] WKND SPA React - UI apps ...........................
SUCCESS [  8.237 s]
[INFO] WKND SPA React - UI content ........................
SUCCESS [  5.633 s]
[INFO] WKND SPA React - UI config .........................
SUCCESS [  0.234 s]
[INFO] WKND SPA React - All ...............................
SUCCESS [  0.643 s]
[INFO] WKND SPA React - Integration Tests .................
SUCCESS [ 12.377 s]
[INFO] WKND SPA React - Dispatcher ........................
SUCCESS [  0.066 s]
[INFO] WKND SPA React - UI Tests ..........................
SUCCESS [  0.074 s]
[INFO] WKND SPA React - Project Analyser ..................
SUCCESS [ 31.287 s]
[INFO] ------------------------------------------------------------
-------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
-------------
```

The Maven profile `autoInstallSinglePackage` compiles the individual modules of the project and deploys a single package to the AEM instance. By default this package will be deployed to an AEM instance running locally on port **4502** and with the credentials of `admin:admin`.

4. Navigate to **Package Manager** on your local AEM instance: http://localhost:4502/crx/packmgr/index.jsp.

5. You should see multiple packages prefixed with `aem-guides-wknd-spa.react`.



AEM Package Manager

All of the custom code needed for the project will be bundled into these packages and installed on the AEM environment.

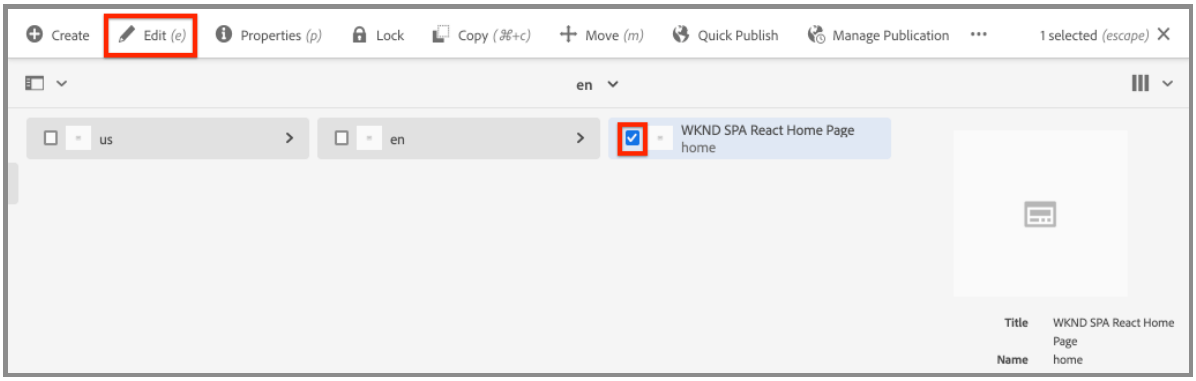
Author Content

Next, open the starter SPA that was generated by the archetype and update some of the content.

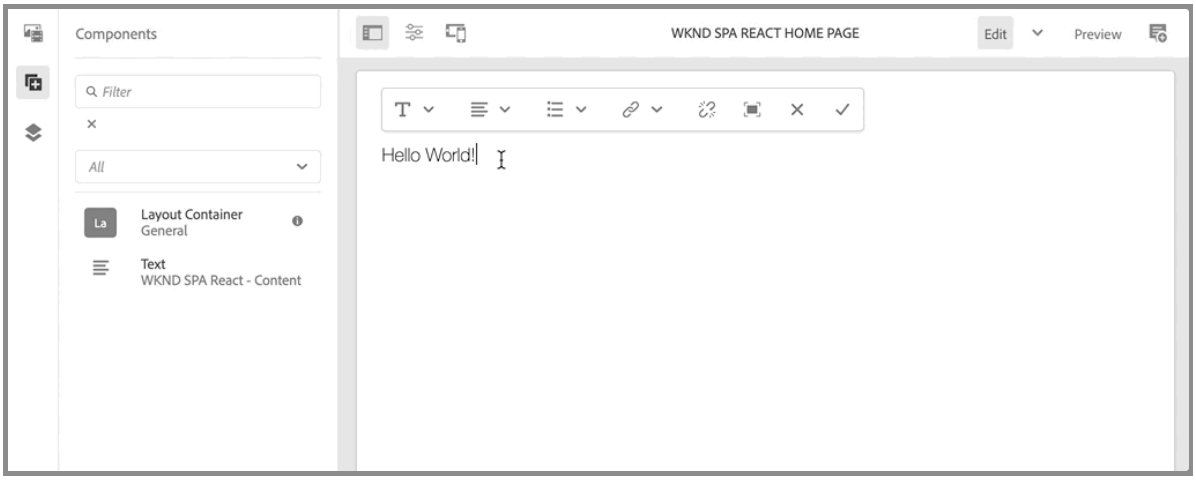
1. Navigate to the **Sites** console: http://localhost:4502/sites.html/content.

   The WKND SPA includes a basic site structure with a country, language and home page. This hierarchy is based on the archetype's default values for `language_country` and `isSingleCountryWebsite`. These values can be overwritten by updating the available properties when generating a project.

2. Open the **us** > **en** > **WKND SPA React Home Page** page by selecting the page and clicking the **Edit** button in the menu bar:



3. A **Text** component has already been added to the page. You can edit this component like any other component in AEM.



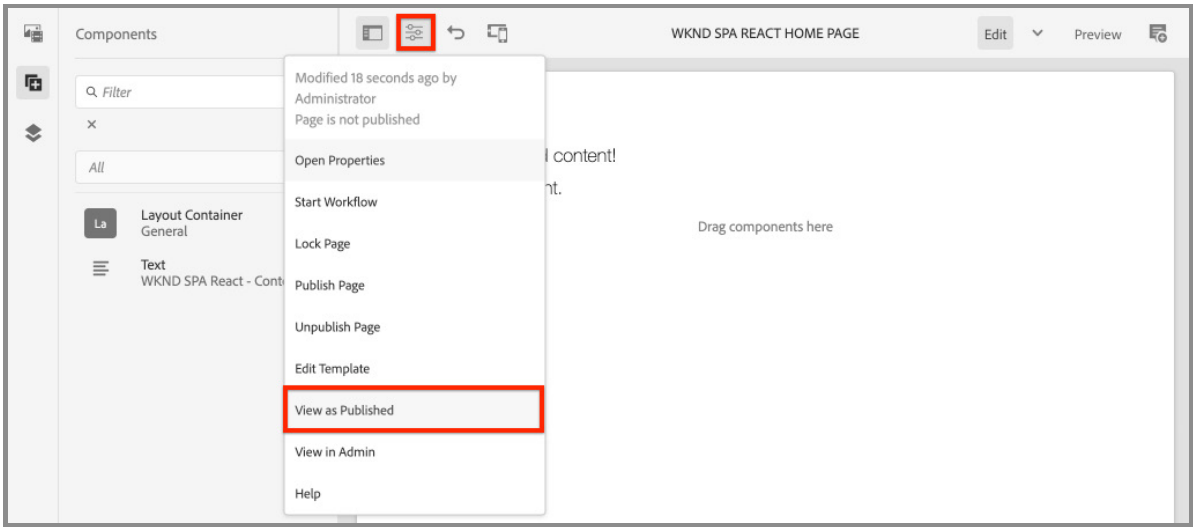
4. Add an additional **Text** component to the page.

   Notice that the authoring experience is similar to that of a traditional AEM Sites page. Currently a limited number of components are available to be used. More will be added over the course of these labs.

Inspect the Single Page Application

Next, verify that this is a Single Page Application with the use of your browser's developer tools.

1. In the **Page Editor**, click the **Page Information** button > **View as Published**:



   This will open a new tab with the query parameter `?wcmmode=disabled` which effectively turns off the AEM editor: http://localhost:4502/content/wknd-spa-react/us/en/home.html?wcmmode=disabled

2. View the page's source and notice that the text content **[Hello World]** or any of the other content is not found. Instead you should see HTML like the following:

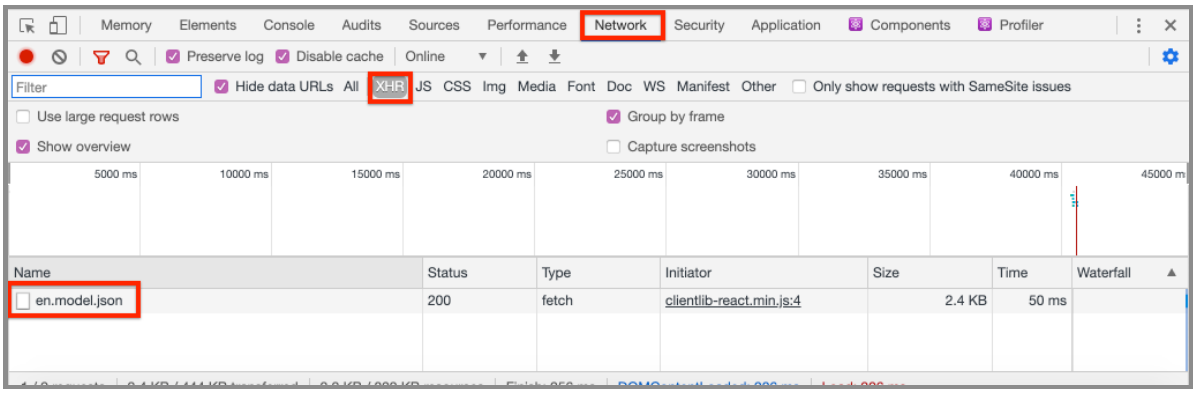
```
...
<body>
    <noscript>You need to enable JavaScript to run this app.
</noscript>
    <div id="spa-root"></div>
    <script type="text/javascript"
src="../../../../../../etc.clientlibs/wknd-spa-
react/clientlibs/clientlib-react.lc-xxxx-lc.min.js"></script>
</body>
...
```

   `clientlib-react.min.js` is the React SPA that is loaded on to the page and responsible for rendering the content.

   However, *where does the content come from?*

3. Return to the tab: http://localhost:4502/content/wknd-spa-react/us/en/home.html?wcmmode=disabled

4. Open the browser's developer tools and inspect the network traffic of the page during a refresh. View the **XHR** requests:



There should be a request to http://localhost:4502/content/wknd-spa-react/us/en.model.json. This contains all of the content, formatted in JSON, that will drive the SPA.

5. In a new tab open http://localhost:4502/content/wknd-spa-react/us/en.model.json

   The request `en.model.json` represents the content model that will drive the application. Inspect the JSON output and you should be able to find the snippet representing the **[Text]** component(s).

```
...
":items": {
    "text": {
        "text": "<p>Hello World! Updated content!</p>\r\n",
        "richText": true,
        ":type": "wknd-spa-react/components/text"
    },
    "text_98796435": {
        "text": "<p>A new text component.</p>\r\n",
        "richText": true,
        ":type": "wknd-spa-react/components/text"
    },
}
...
```

   In the next chapter we will inspect how this JSON content is mapped from AEM Components to SPA Components to form the basis of the AEM SPA Editor experience.

NOTE:

It may be helpful to install a browser extension to automatically format the JSON output.

Congratulations, you have just created your first AEM SPA Editor Project!

The SPA is quite simple. In the next few chapters more functionality will be added.

Integrate the SPA

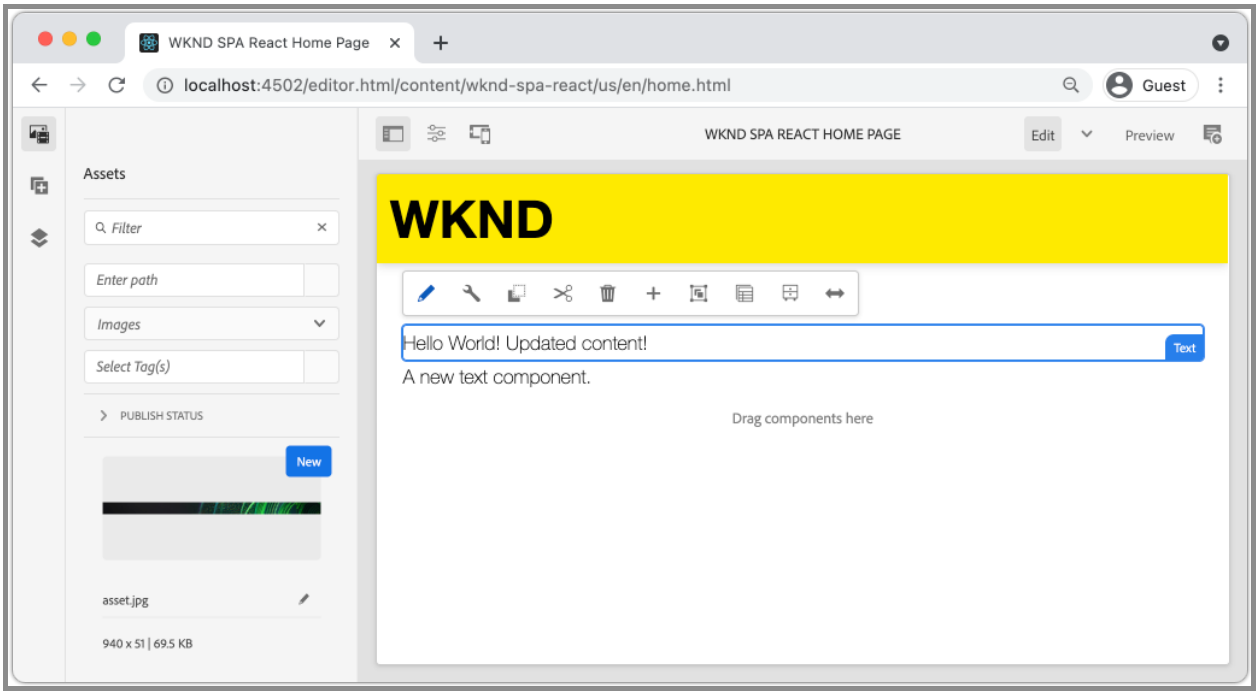
Understand how the source code for a Single Page Application (SPA) written in React can be integrated with an Adobe Experience Manager (AEM) Project. Learn to use modern front-end tools, like a webpack dev server, to rapidly develop the SPA against the AEM JSON model API.

Objectives

1. Understand how the SPA project is integrated with AEM with client-side libraries.
2. Learn how to use a webpack development server for dedicated front-end development.
3. Explore the use of a **proxy** and static **mock** file for developing against the AEM JSON model API.

What You Will Build

In this chapter you will make several small changes to the SPA in order to understand how it is integrated with AEM.
This chapter will add a simple `Header` component to the SPA. In the process of building out this **static** `Header` component several approaches to AEM SPA development will be used.



The SPA is extended to add a static `Header` component

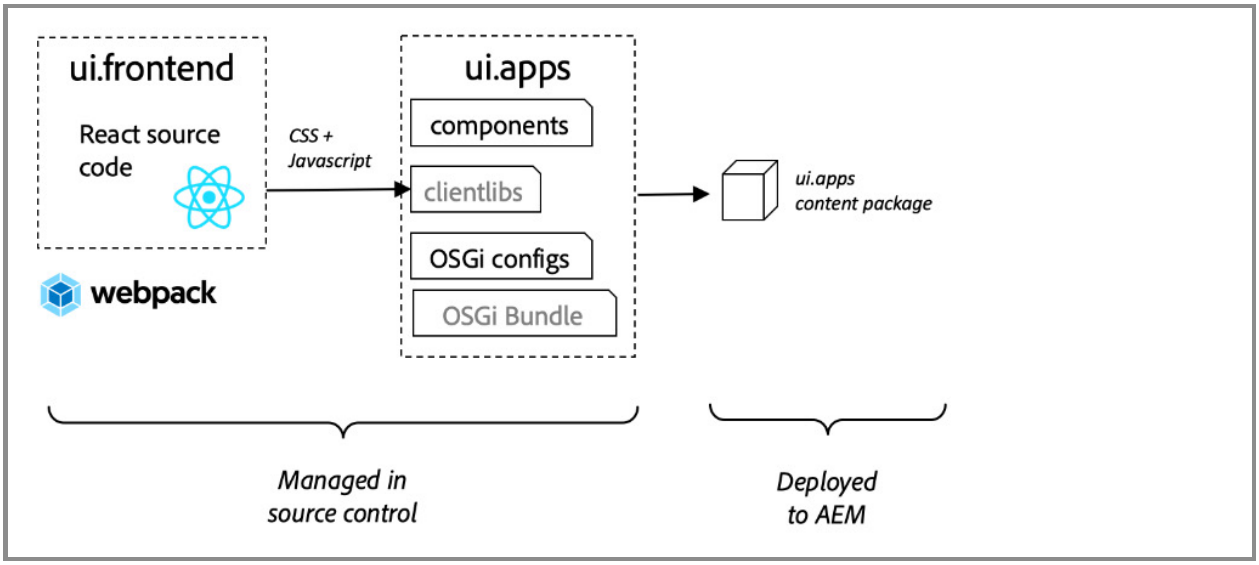
Prerequisites

Review the required tooling and instructions for setting up a local development environment. This chapter is a continuation of the Create Project chapter, however to follow along all you need is a working SPA-enabled AEM project.

Integration Approach

Two modules were created as part of the AEM project: `ui.apps` and `ui.frontend`.

The `ui.frontend` module is a webpack project that contains all of the SPA source code. A majority of the SPA development and testing will be done in the webpack project. When a production build is triggered, the SPA is built and compiled using webpack. The compiled artifacts (CSS and Javascript) are copied into the `ui.apps` module which is then deployed to the AEM runtime.



A high-level depiction of the SPA integration.

Additional information about the Front-end build can be found here.

Inspect the SPA integration

Next, inspect the `ui.frontend` module to understand the SPA that has been auto-generated by the AEM Project archetype.

1. In the IDE of your choice open your AEM Project. This course will use the Visual Studio Code IDE.

2. Expand and inspect the `ui.frontend` folder. Open the file `ui.frontend/package.json`

3. Under the `dependencies` you should see several related to `react` including `react-scripts`

   The `ui.frontend` is a React application based on the Create React App or CRA for short. The `react-scripts` version indicates which version of CRA is used.

4. There are also several dependencies prefixed with `@adobe`:

   ```
   "@adobe/aem-react-editable-components": "~1.1.2",
   "@adobe/aem-spa-component-mapping": "~1.1.0",
   "@adobe/aem-spa-page-model-manager": "~1.3.3",
   "@adobe/aem-core-components-react-base": "1.1.8",
   "@adobe/aem-core-components-react-spa": "1.1.7",
   ```

   The above modules make up the AEM SPA Editor JS SDK and provide the functionality to make it possible to map SPA Components to AEM Components.

   Also included are AEM WCM Components - React Core implementation and AEM WCM Components - Spa editor - React Core implementation. These are a set of re-usable UI components that map to out of the box AEM components. These are designed to be used as is and styled to meet your project's needs.

5. In the `package.json` file there are several `scripts` defined:

```
"scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build && clientlib",
    "test": "react-scripts test",
    "eject": "react-scripts eject",
}
```

   These are standard build scripts made available by the Create React App.

   The only difference is the addition of `&& clientlib` to the `build` script. This extra instruction is responsible for copying the compiled SPA into the `ui.apps` module as a client-side library during a build.

   The npm module aem-clientlib-generator is used to facilitate this.

6. Inspect the file `ui.frontend/clientlib.config.js`. This configuration file is used by aem-clientlib-generator to determine how to generate the client library.

7. Inspect the file `ui.frontend/pom.xml`. This file transforms the `ui.frontend` folder into a Maven module. The `pom.xml` file has been updated to use the frontend-maven-plugin to **test** and **build** the SPA during a Maven build.

8. Inspect the file `index.js` at `ui.frontend/src/index.js`:

```
//ui.frontend/src/index.js
...
document.addEventListener('DOMContentLoaded', () => {
    ModelManager.initialize().then(pageModel => {
        const history = createBrowserHistory();
        render(
        <Router history={history}>
            <App
            history={history}
            cqChildren={pageModel[Constants.CHILDREN_PROP]}
            cqItems={pageModel[Constants.ITEMS_PROP]}
            cqItemsOrder={pageModel[Constants.ITEMS_ORDER_PROP]}
            cqPath={pageModel[Constants.PATH_PROP]}
            locationPathname={window.location.pathname}
            />
        </Router>,
        document.getElementById('spa-root')
        );
    });
});
```

`index.js` is the entrypoint of the SPA. `ModelManager` is provided by the AEM SPA Editor JS SDK. It is responsible for calling and injecting the `pageModel` (the JSON content) into the application.

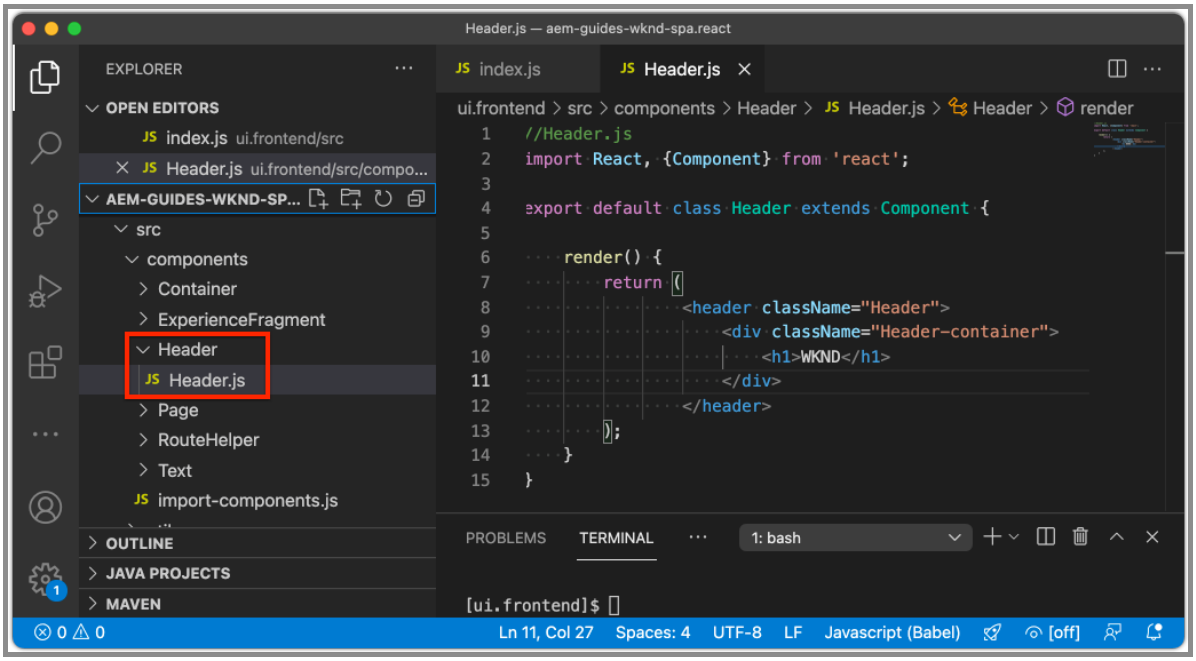
9. Inspect the file `import-component.js` at `ui.frontend/src/import-components.js`. This file imports the out of the box **React Core Components** and makes them available to the project. We will inspect the mapping of AEM content to SPA components in the next chapter.

Add a Static SPA Component

Next, add a new component to the SPA and deploy the changes to a local AEM instance. This will be a simple change, just to illustrate how the SPA is updated.

1. In the `ui.frontend` module, beneath `ui.frontend/src/components` create a new folder named `Header`.

2. Create a file named `Header.js` beneath the `Header` folder.



3. Populate `Header.js` with the following:

```
//Header.js
import React, {Component} from 'react';

export default class Header extends Component {

    render() {
        return (
                <header className="Header">
                    <div className="Header-container">
                        <h1>WKND</h1>
                    </div>
                </header>
        );
    }
}
```

Above is a standard React component that will output a static text string.

4. Open the file `ui.frontend/src/App.js`. This is the application entry-point.

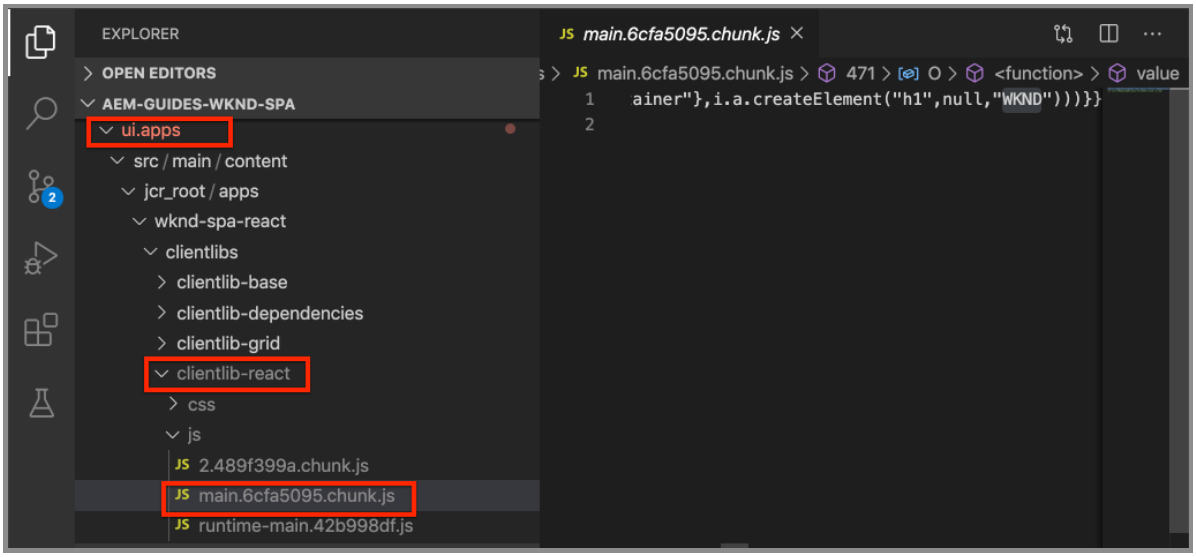
5. Make the following updates to `App.js` to include the static `Header`:

```
  import { Page, withModel } from '@adobe/aem-react-editable-
components';
  import React from 'react';
+ import Header from './components/Header/Header';

  // This component is the application entry point
  class App extends Page {
  render() {
      return (
      <div>
+       <Header />
         {this.childComponents}
         {this.childPages}
     </div>
```

6. Open a new terminal and navigate into the `ui.frontend` folder and run the `npm run build` command:

```
$ cd aem-guides-wknd-spa
$ cd ui.frontend
$ npm run build
...
Compiled successfully.

File sizes after gzip:

118.95 KB (-33 B)  build/static/js/2.489f399a.chunk.js
1.11 KB (+48 B)    build/static/js/main.6cfa5095.chunk.js
806 B              build/static/js/runtime-main.42b998df.js
451 B              build/static/css/main.e57bbe8a.chunk.css
```

7. Navigate to the `ui.apps` folder. Beneath `ui.apps/src/main/content/jcr_root/apps/wknd-spa-react/clientlibs/clientlib-react` you should see the compiled SPA files have been copied from the `ui.frontend/build` folder.

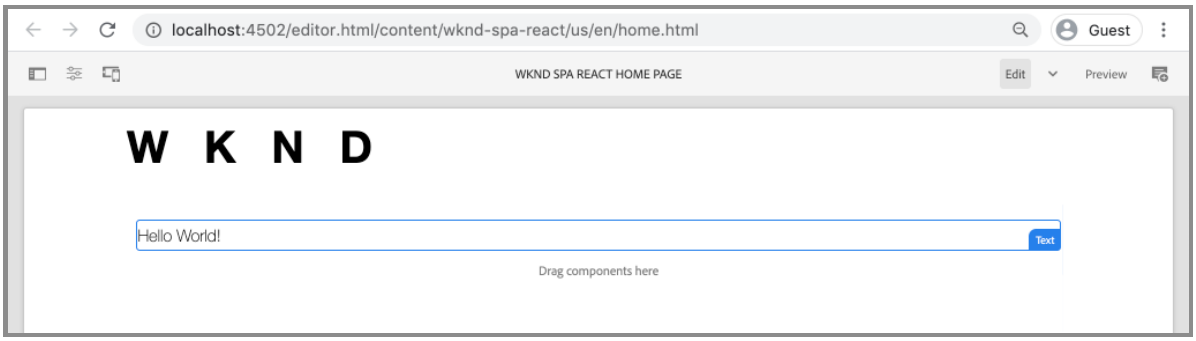


8. Return to the terminal and navigate into the `ui.apps` folder. Execute the following Maven command:

```
$ cd ../ui.apps
$ mvn clean install -PautoInstallPackage
...
[INFO] ------------------------------------------------------------
--------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
--------------
[INFO] Total time:  9.629 s
[INFO] Finished at: 2020-05-04T17:48:07-07:00
[INFO] ------------------------------------------------------------
--------------
```

   This will deploy the `ui.apps` package to a local running instance of AEM.

9. Open a browser tab and navigate to http://localhost:4502/editor.html/content/wknd-spa-react/us/en/home.html. You should now see the contents of the `Header` component being displayed in the SPA.



   The above steps are executed automatically when triggering a Maven build from the root of the project (i.e `mvn clean install -PautoInstallSinglePackage`). You should now understand the basics of the integration between the SPA and AEM client-side libraries. Notice that you can still edit and add `Text` components in AEM beneath the static `Header` component.

Webpack Dev Server - Proxy the JSON API

As seen in the previous exercises, performing a build and syncing the client library to a local instance of AEM takes a few minutes. This is acceptable for final testing, but is not ideal for the majority of the SPA development.

A webpack-dev-server can be used to rapidly develop the SPA. The SPA is driven by a JSON model generated by AEM. In this exercise the JSON content from a running instance of AEM will be **proxied** into the development server.

1. Return to the IDE and open the file `ui.frontend/package.json`.

   Look for a line like the following:

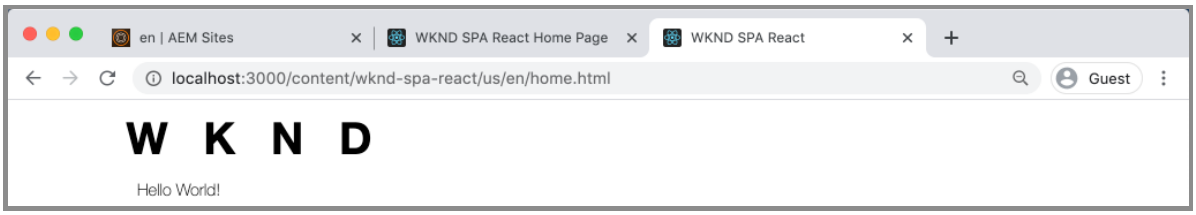
   ```
   "proxy": "http://localhost:4502",
   ```

   The Create React App provides an easy mechanism to proxy API requests. All unknown requests will be proxied through `localhost:4502`, the local AEM quickstart.

2. Open a terminal window and navigate to the `ui.frontend` folder. Run the command `npm start`:

   ```
   $ cd ui.frontend
   $ npm start
   ...
   Compiled successfully!

   You can now view wknd-spa-react in the browser.

   Local:            http://localhost:3000
   On Your Network:  http://192.168.86.136:3000

   Note that the development build is not optimized.
   To create a production build, use npm run build.
   ```

3. Open a new browser tab (if not already opened) and navigate to http://localhost:3000/content/wknd-spa-react/us/en/home.html.



You should see the same content as in AEM, but without any of the authoring capabilities enabled.

NOTE:

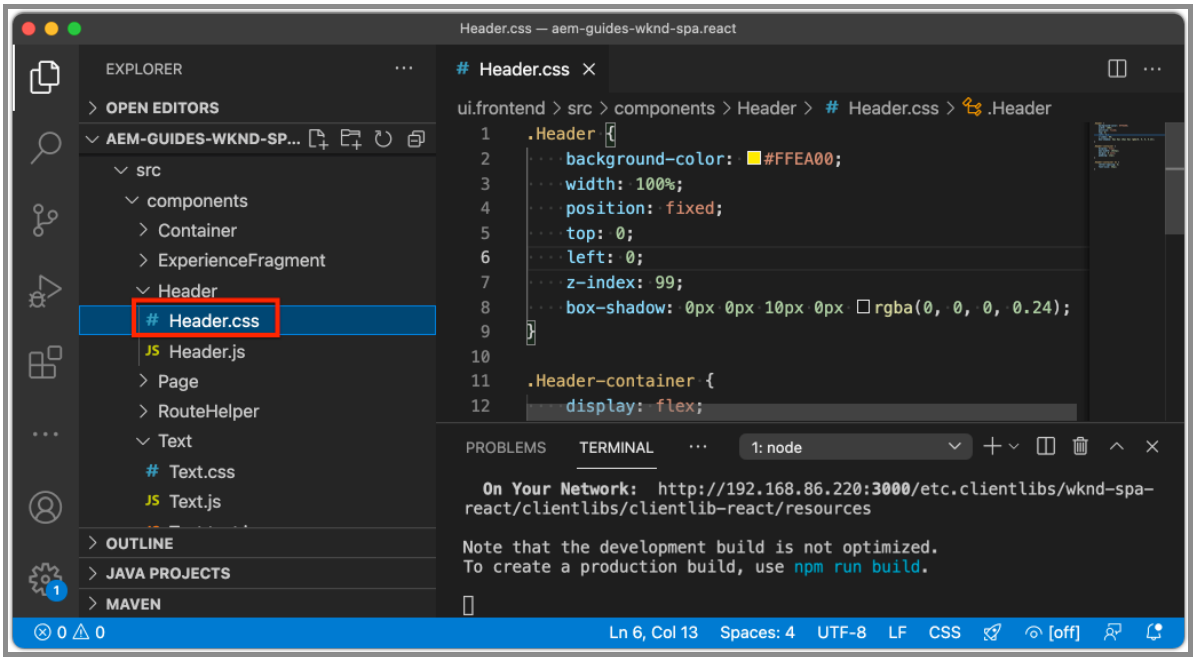
Due to the security requirements of AEM, you will need to be logged into the local AEM instance (http://localhost:4502) in the same browser but in a different tab.

---

4. Return to the IDE and create a file named `Header.css` in the `src/components/Header` folder.

5. Populate the `Header.css` with the following:

```css
.Header {
    background-color: #FFEA00;
    width: 100%;
    position: fixed;
    top: 0;
    left: 0;
    z-index: 99;
    box-shadow: 0px 0px 10px 0px rgba(0, 0, 0, 0.24);
}

.Header-container {
    display: flex;
    max-width: 1024px;
    margin: 0 auto;
    padding: 12px;
}

.Header-container h1 {
    letter-spacing: 0;
    font-size: 48px;
}
```



6. Re-open `Header.js` and add the following line to reference `Header.css`:

```
  //Header.js
  import React, {Component} from 'react';
+ require('./Header.css');
```

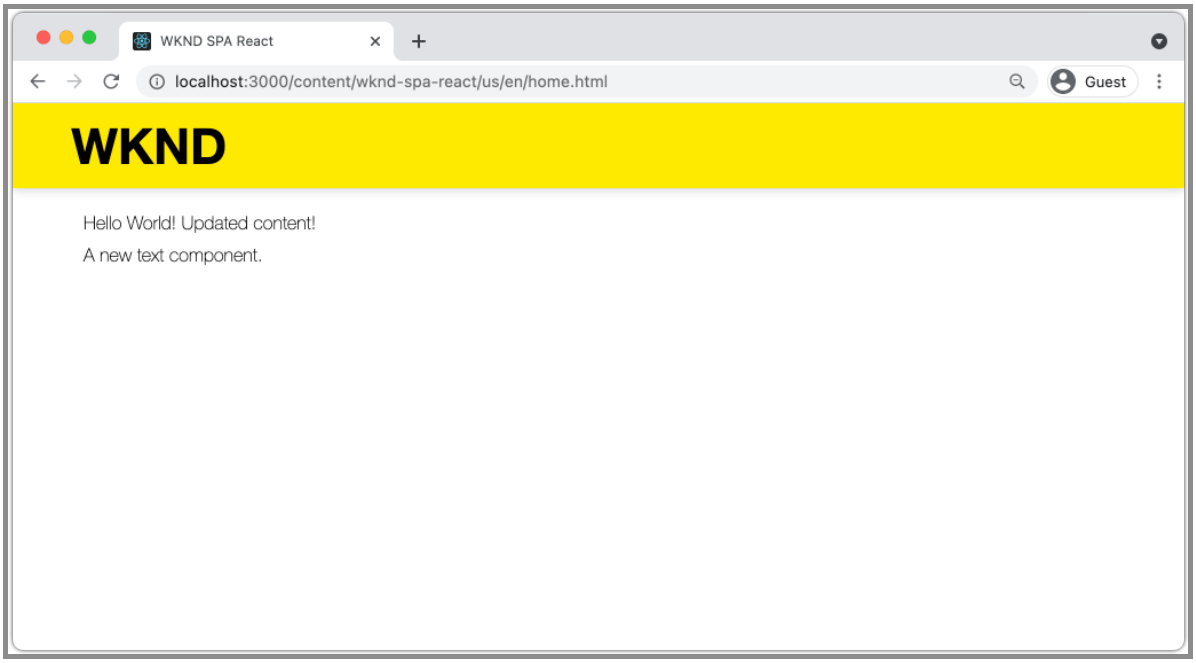
   Save the changes.

7. Navigate to http://localhost:3000/content/wknd-spa-react/us/en/home.html to see the style changes automatically reflected.

8. Open the file `Page.css` at `ui.frontend/src/components/Page`. Make the following changes to fix the padding:

```
.page {
  max-width: 1024px;
  margin: 0 auto;
  padding: 12px;
  padding-top: 50px;
}
```

9. Return to the browser at http://localhost:3000/content/wknd-spa-react/us/en/home.html. You should immediately see the changes to the app reflected.



   You can continue to make content updates in AEM and see them reflected in **webpack-dev-server**, since we are proxying the content.

10. Stop the webpack dev server with `ctrl+c` in the terminal.

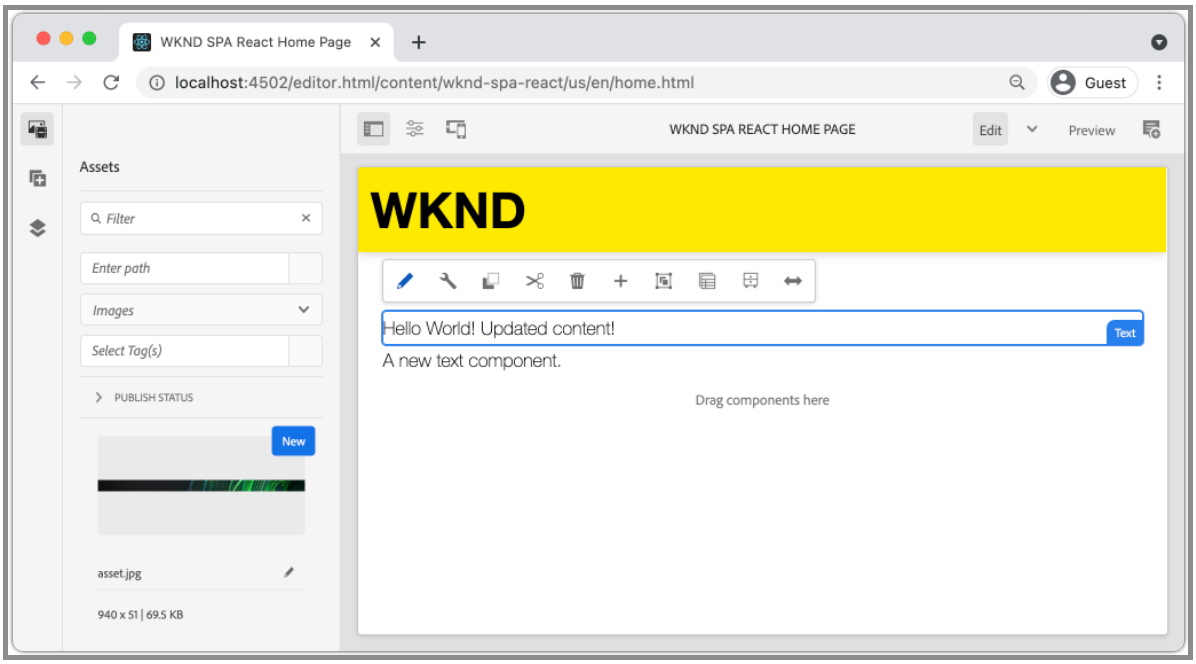
Deploy SPA Updates to AEM

The changes made to the `Header` are currently only visible through the **webpack-dev-server**. Deploy the updated SPA to AEM to see the changes.

1. Navigate to the root of the project (`aem-guides-wknd-spa`) and deploy the project to AEM using Maven:

   ```
   $ cd ..
   $ mvn clean install -PautoInstallSinglePackage
   ```

2. Navigate to http://localhost:4502/editor.html/content/wknd-spa-react/us/en/home.html. You should see the updated `Header` and styles applied.

   Now that the updated SPA is in AEM, authoring can continue.

Congratulations, you have updated the SPA and explored the integration with AEM! You know know how to develop the SPA against the AEM JSON model API using a **webpack-dev-server**.

(Bonus) Webpack Dev Server - Mock JSON API

Another approach to rapid development is to use a static JSON file to act as the JSON model. By "mocking" the JSON, we remove the dependency on a local AEM instance. It also allows a front-end developer to update the JSON model in order to test functionality and drive changes to the JSON API that would then be later implemented by a back-end developer.

The initial set up of the mock JSON does **require a local AEM instance**.

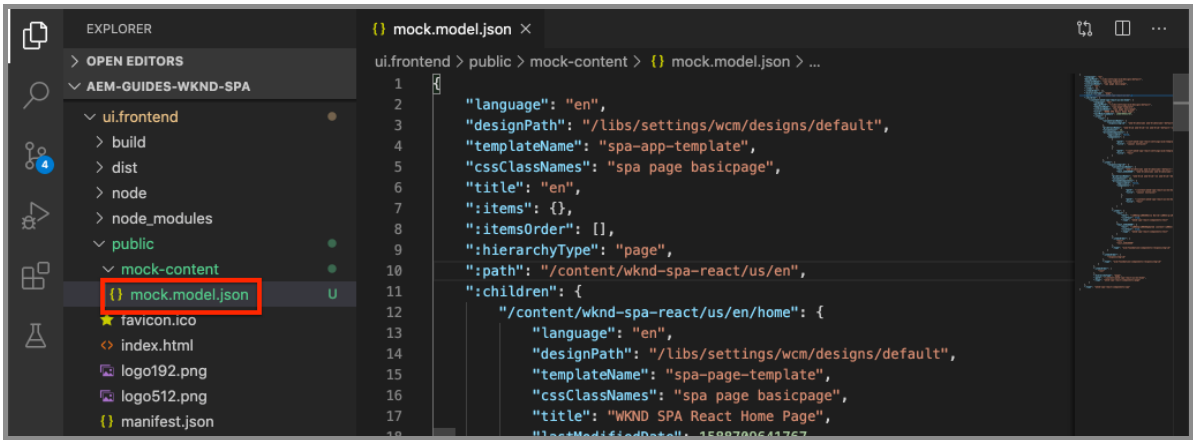
1. Return to the IDE and navigate to `ui.frontend/public` and add a new folder named `mock-content`.

2. Create a new file named `mock.model.json` beneath `ui.frontend/public/mock-content`.

3. In the browser navigate to http://localhost:4502/content/wknd-spa-react/us/en.model.json.

   This is the JSON exported by AEM that is driving the application. Copy the JSON output.

4. Paste the JSON output from the previous step in the file `mock.model.json`.



5. Open the file `index.html` at `ui.frontend/public/index.html`. Update the metadata property for the AEM page model to point to a variable `%REACT_APP_PAGE_MODEL_PATH%`:

```
<!-- AEM page model -->
<meta
   property="cq:pagemodel_root_url"
   content="%REACT_APP_PAGE_MODEL_PATH%"
/>
```

   Using a variable for the value of the `cq:pagemodel_root_url` will make it easier to toggle between the proxy and mock json model.

6. Open the file `ui.frontend/.env.development` and make the following updates to comment out the previous value for `REACT_APP_PAGE_MODEL_PATH`:

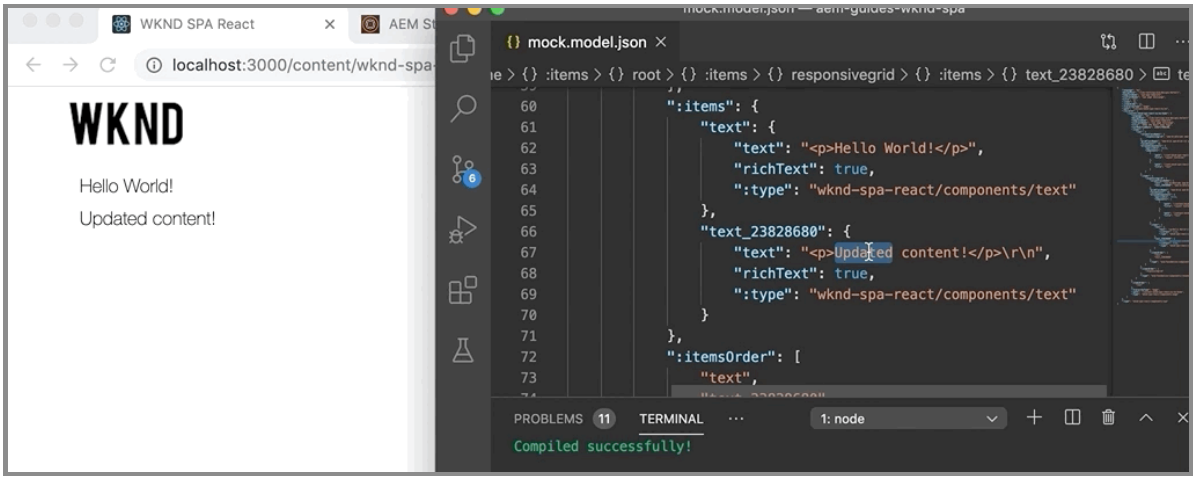
```
+ PUBLIC_URL=/
- PUBLIC_URL=/etc.clientlibs/wknd-spa-react/clientlibs/clientlib-react/resources

- REACT_APP_PAGE_MODEL_PATH=/content/wknd-spa-react/us/en.model.json
+ REACT_APP_PAGE_MODEL_PATH=/mock-content/mock.model.json

REACT_APP_ROOT=/content/wknd-spa-react/us/en/home.html
```

7. If currently running, stop the **webpack-dev-server**. Start the **webpack-dev-server** from the terminal:

```
$ cd ui.frontend
$ npm start
```

   Navigate to http://localhost:3000/content/wknd-spa-react/us/en/home.html and you should see the SPA with the same content used in the **proxy** json.

8. Make a small change to the `mock.model.json` file created earlier. You should see the updated content immediately reflected in the **webpack-dev-server**.



Being able to manipulate the JSON model and see the effects on a live SPA can help a developer understand the JSON model API. It also allows both front-end and back-end development happen in parallel.

You can now toggle where to consume the JSON content by toggling the entries in the `env.development` file:

```
# JSON API via proxy to AEM
#REACT_APP_PAGE_MODEL_PATH=/content/wknd-spa-react/us/en.model.json

# JSON API via static mock file
REACT_APP_PAGE_MODEL_PATH=/mock-content/mock.model.json
```

Map SPA Components to AEM Components

Learn how to map React components to Adobe Experience Manager (AEM) components with the AEM SPA Editor JS SDK. Component mapping enables users to make dynamic updates to SPA components within the AEM SPA Editor, similar to traditional AEM authoring.

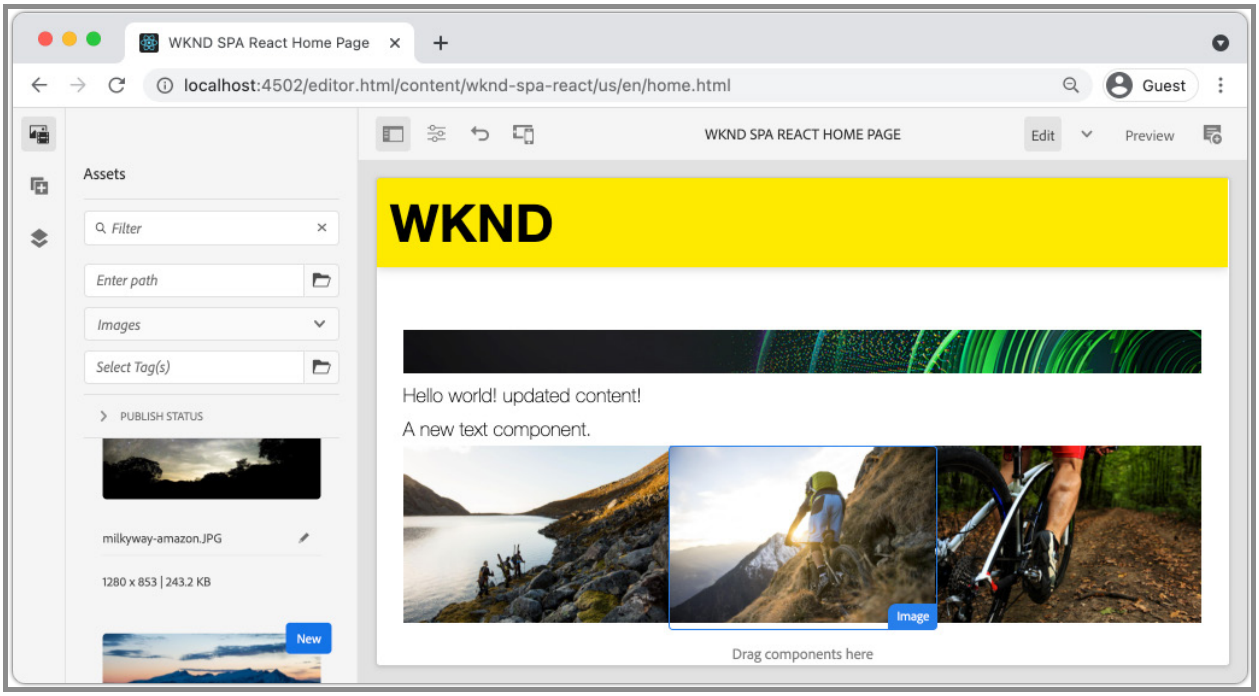
This chapter takes a deeper-dive into the AEM JSON model API and how the JSON content exposed by an AEM component can be automatically injected into a React component as props.

Objectives

1. Learn how to map AEM components to SPA Components.
2. Inspect how a React component uses dynamic properties passed from AEM.
3. Learn how to use out of the box React AEM Core Components.

What You Will Build

This chapter will inspect how the provided `Text` SPA component is mapped to the AEM `Text` component. React Core Components like the `Image` SPA component will be used in the SPA and authored in AEM. Out of the box features of the **Layout Container** and **Template Editor** policies will also be used to create a view that is a little more varied in appearance.

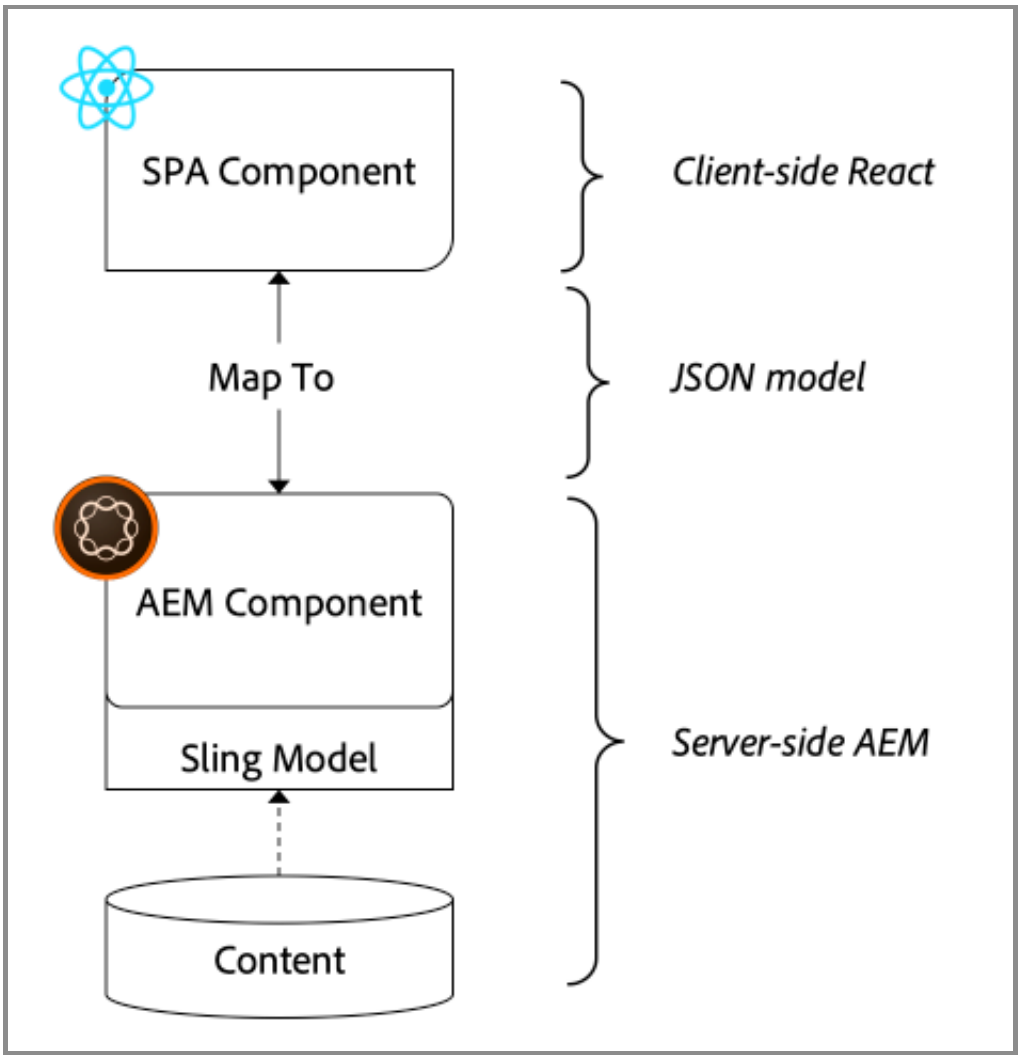


Prerequisites

Review the required tooling and instructions for setting up a local development environment. This chapter is a continuation of the Integrate the SPA chapter, however to follow along all you need is a SPA-enabled AEM project.

Mapping Approach

The basic concept is to map a SPA Component to an AEM Component. AEM components, run server-side, export content as part of the JSON model API. The JSON content is consumed by the SPA, running client-side in the browser. A 1:1 mapping between SPA components and an AEM component is created.



High-level overview of mapping an AEM Component to a React Component

Inspect the Text Component

The AEM Project Archetype provides a `Text` component that is mapped to the AEM Text component. This is an example of a **content** component, in that it renders *content* from AEM.

Let's see how the component works.

Inspect the JSON Model

1. Before jumping into the SPA code, it is important to understand the JSON model that AEM provides. Navigate to the Core Component Library and view the page for the Text component. The Core Component Library provides examples of all the AEM Core Components.

2. Select the **JSON** tab for one of the examples:

   You should see three properties: `text`, `richText`, and `:type`.

   `:type` is a reserved property that lists the `sling:resourceType` (or path) of the AEM Component. The value of `:type` is what is used to map the AEM component to the SPA component.

   `text` and `richText` are additional properties that will be exposed to the SPA component.

3. View the JSON output at http://localhost:4502/content/wknd-spa-react/us/en.model.json. You should be able to find an entry similar to:

```
"text": {
    "id": "text-a647cec03a",
    "text": "<p>Hello World! Updated content!</p>\r\n",
    "richText": true,
    ":type": "wknd-spa-react/components/text",
    "dataLayer": {}
  }
```

Inspect the Text SPA C/omponent

1. In the IDE of your choice open up the AEM Project for the SPA. Expand the `ui.frontend` module and open the file `Text.js` under `ui.frontend/src/components/Text/Text.js`.

2. The first area we will inspect is the `class Text` at ~line 40:

```
class Text extends Component {

    get richTextContent() {
        return (<div
                id={extractModelId(this.props.cqPath)}
                data-rte-editelement
                dangerouslySetInnerHTML={{__html:
DOMPurify.sanitize(this.props.text)}} />
                );
    }

    get textContent() {
        return <div>{this.props.text}</div>;
    }

    render() {
        return this.props.richText ? this.richTextContent :
this.textContent;
    }
}
```

`Text` is a standard React component. The component uses `this.props.richText` to determine whether the content to render is going to be rich text or plain text. The actual "content" used comes from `this.props.text`.

To avoid a potential XSS attack, the rich text is escaped via `DOMPurify` before using dangerouslySetInnerHTML to render the content. Recall the `richText` and `text` properties from the JSON model earlier in the exercise.

3. Next take a look at the `TextEditConfig` at ~line 29:

```
const TextEditConfig = {
emptyLabel: 'Text',

    isEmpty: function(props) {
        return !props || !props.text || props.text.trim().length
< 1;
    }
};
```

The above code is responsible for determining when to render the placeholder in the AEM author environment. If the `isEmpty` method returns **true** then the placeholder will be rendered.

4. Finally take a look at the `MapTo` call at ~line 62:

```
export default MapTo('wknd-spa-react/components/text')(Text,
TextEditConfig);
```

`MapTo` is provided by the AEM SPA Editor JS SDK (`@adobe/aem-react-editable-components`). The path `wknd-spa-react/components/text` represents the `sling:resourceType` of the AEM component. This path gets matched with the `:type` exposed by the JSON model observed earlier. `MapTo` takes care of parsing the JSON model response and passing the correct values as `props` to the SPA component.

You can find the AEM `Text` component definition at `ui.apps/src/main/content/jcr_root/apps/wknd-spa-react/components/text`.

Use React Core Components

AEM WCM Components - React Core implementation and AEM WCM Components - Spa editor - React Core implementation. These are a set of re-usable UI components that map to out of the box AEM components. Most projects can re-use these components as a starting point for their own implementation.

1. In the project code open the file `import-components.js` at `ui.frontend/src/components`.
   This file imports all of the SPA components that map to AEM components. Given the dynamic nature of the SPA Editor implementation, we must explicitly reference any SPA components that are tied to AEM author-able components. This allows an AEM author to choose to use a component wherever they want in the application.

2. The following import statements include SPA components written in the project:

   ```
   import './Page/Page';
   import './Text/Text';
   import './Container/Container';
   import './ExperienceFragment/ExperienceFragment';
   ```

3. There are several other `imports` from `@adobe/aem-core-components-react-spa` and `@adobe/aem-core-components-react-base`. These are importing the React Core components and making them available in the current project. These are then mapped to project specific AEM components using the `MapTo`, just like with the `Text` component example earlier.

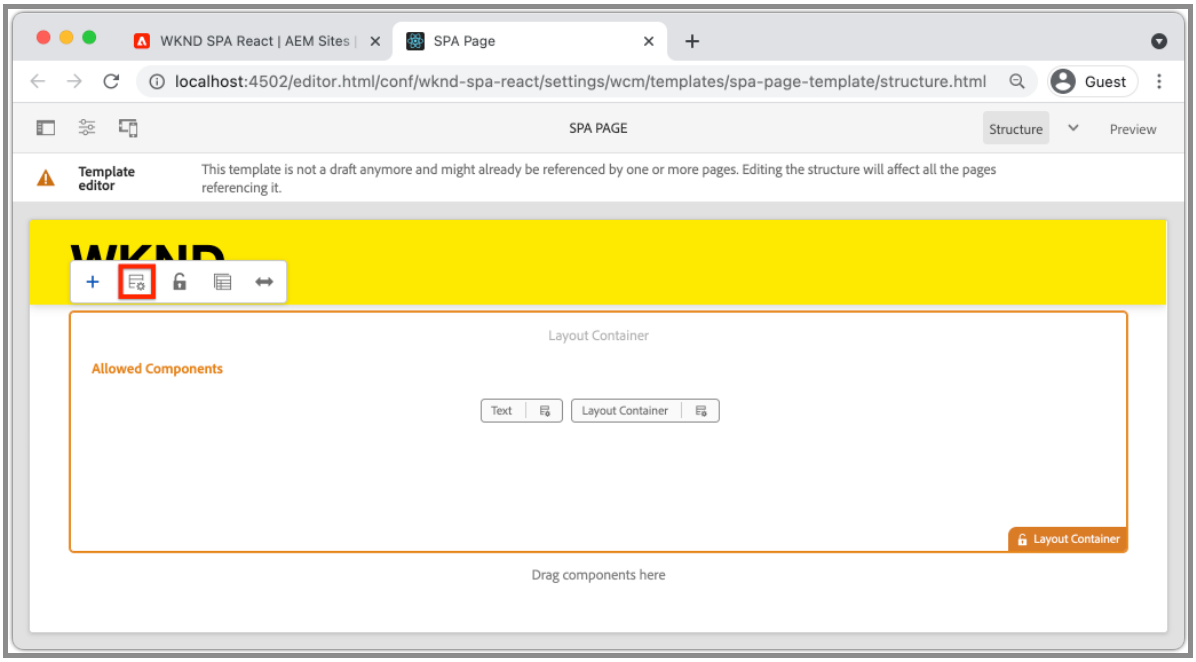
Update AEM Policies

Policies are a feature of AEM templates gives developers and power-users granular control over which components are available to be used. The React Core Components are included in the SPA Code but need to be enabled via a policy before they can be used in the application.

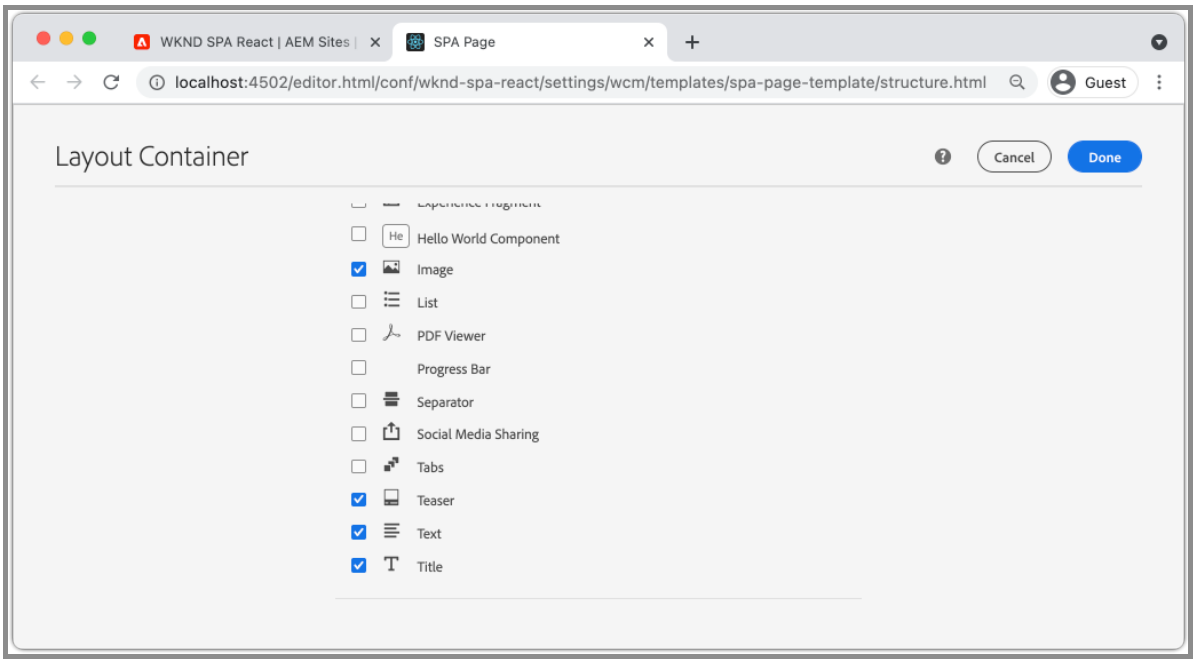
1. From the AEM Start screen navigate to **Tools** > **Templates** > **WKND SPA React**.

2. Select and open the **SPA Page** template for editing.

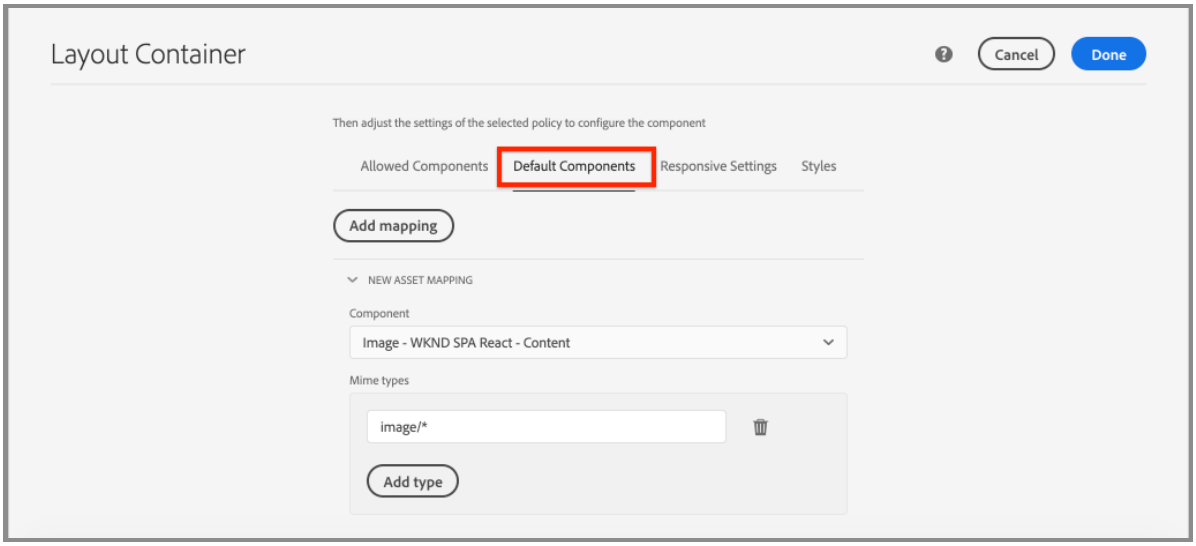
3. Select the **Layout Container** and click it's **policy** icon to edit the policy:



4. Under **Allowed Components** > **WKND SPA React - Content** > check **Image**, **Teaser**, and **Title**.



Under **Default Components** > **Add mapping** and choose the **Image - WKND SPA React - Content** component:

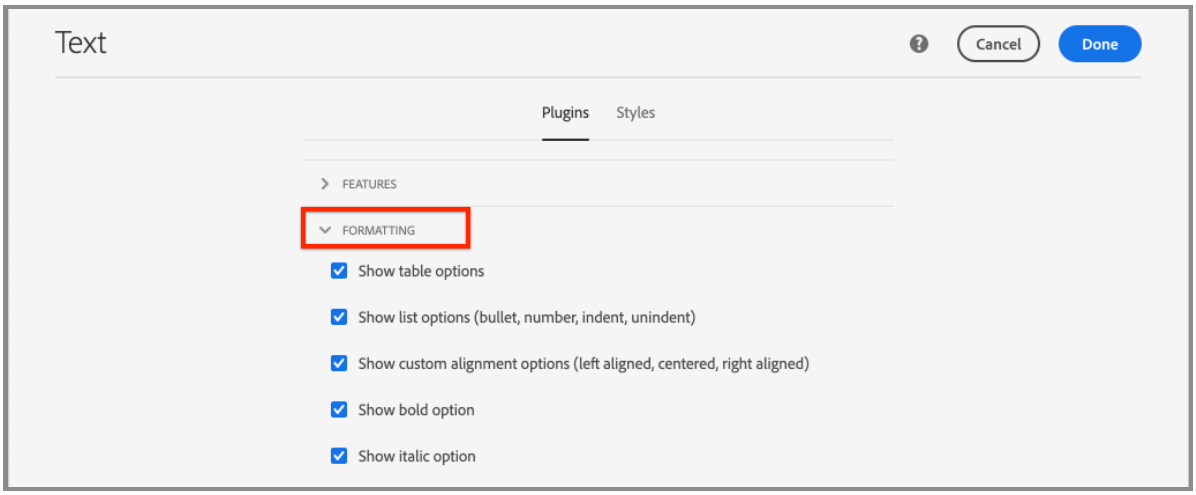


Enter a **mime type** of `image/*`.

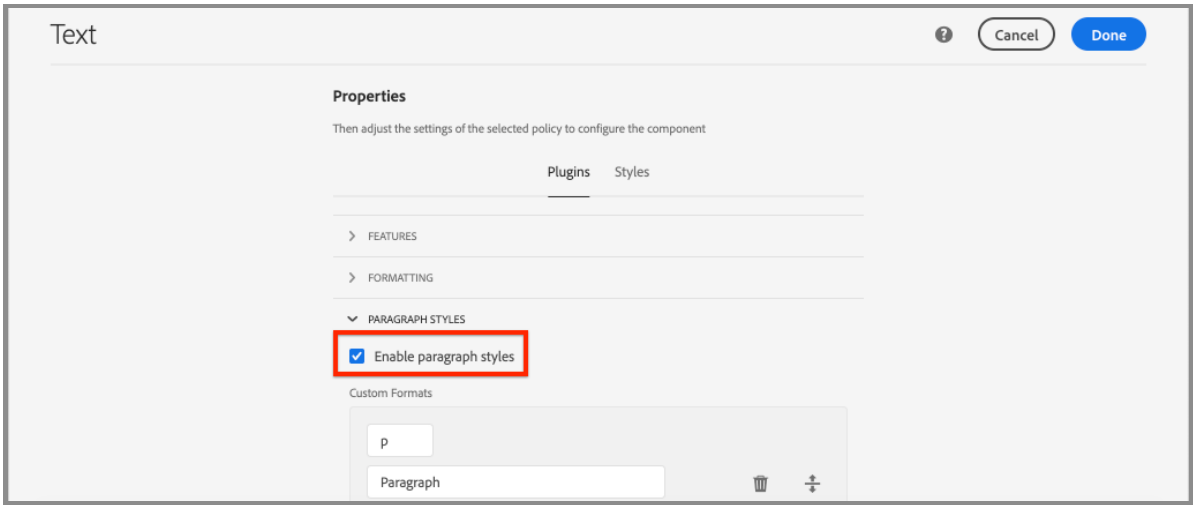
Click **Done** to save the policy updates.

5. In the **Layout Container** click the **policy** icon for the **Text** component.

   Create a new policy named **WKND SPA Text**. Under **Plugins** > **Formatting** > check all the boxes to enable additional formatting options:



   Under **Plugins** > **Paragraph Styles** > check the box to **Enable paragraph styles**:

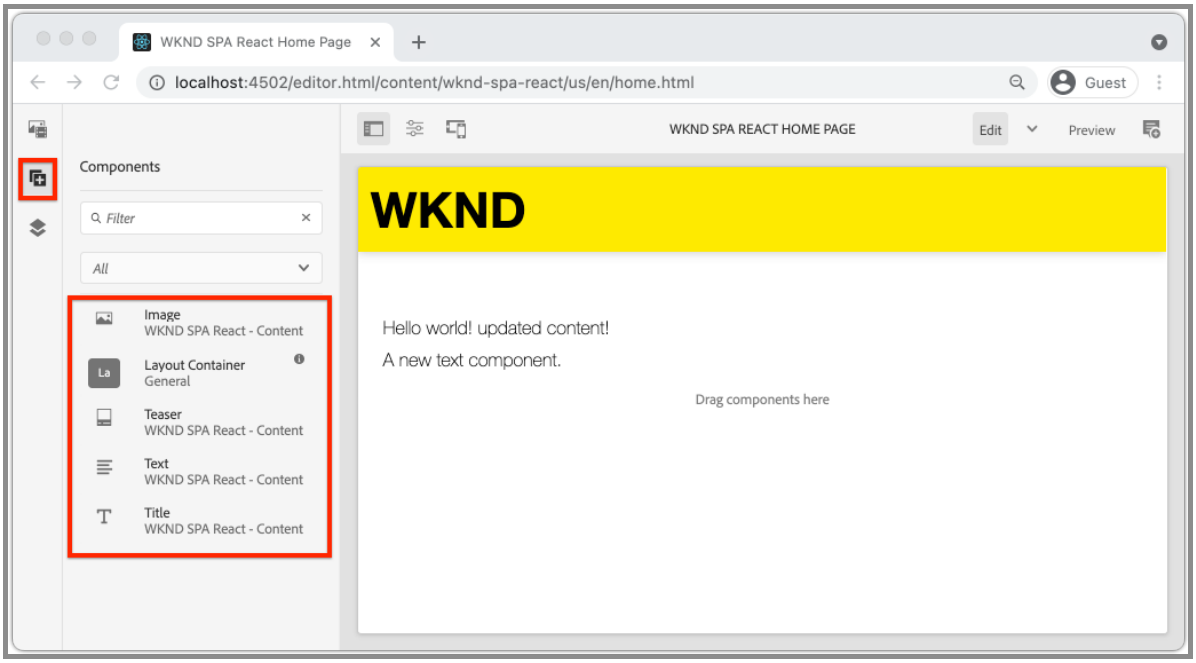


   Click **Done** to save the policy update.

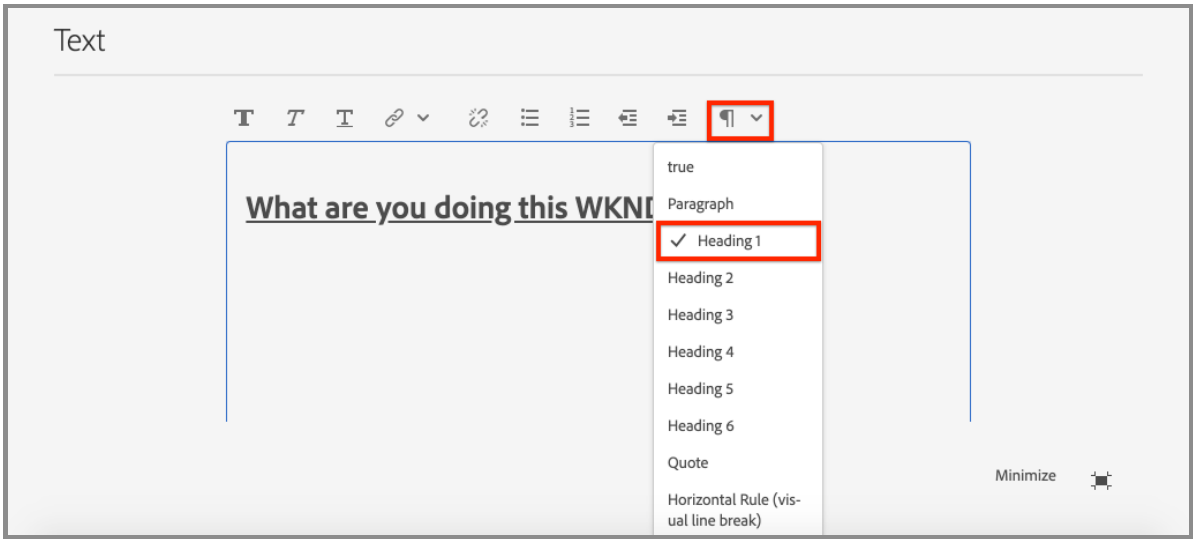
Author Content

1. Navigate to the **Homepage** http://localhost:4502/editor.html/content/wknd-spa-react/us/en/home.html.

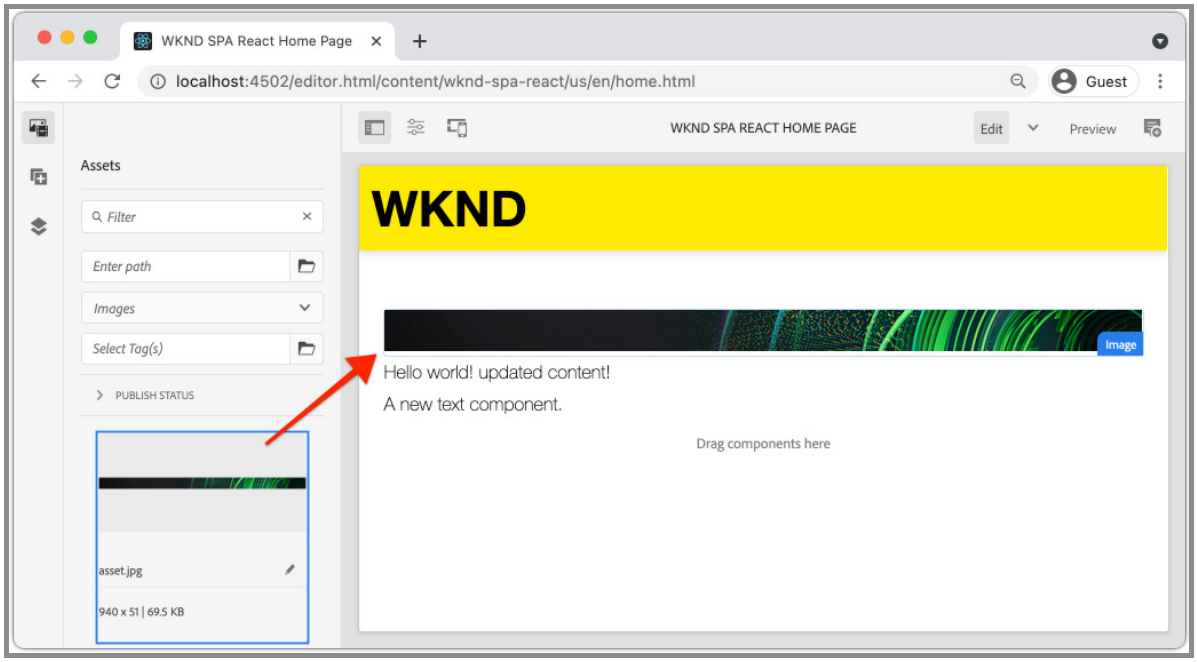
2. You should now be able to use the additional components **Image**, **Teaser**, and **Title** on the page.



3. You should also be able to edit the `Text` component and add additional paragraph styles in **full-screen** mode.

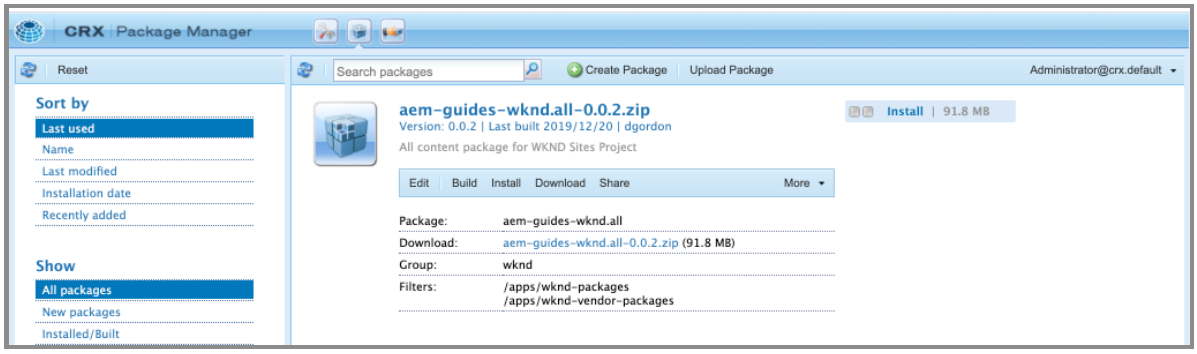


4. You should also be able to drag+drop an image from the **Asset finder**:



5. Experiement with the **Title** and **Teaser** components.

6. Add your own images via AEM Assets or install the finished code base for the standard WKND reference site. The WKND reference site includes many images that can be re-used on the WKND SPA. The package can be installed using AEM's Package Manager.



Inspect the Layout Container

Support for the **Layout Container** is automatically provided by the AEM SPA Editor SDK. The **Layout Container**, as indicated by the name, is a **container** component. Container components are components that accept JSON structures which represent *other* components and dynamically instantiate them.

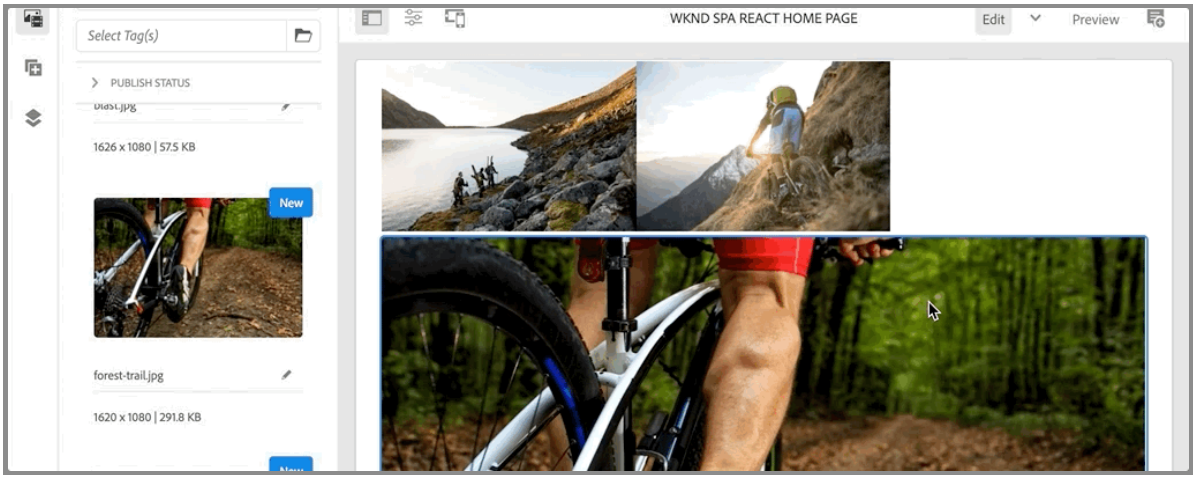
Let's inspect the Layout Container further.

1. In a browser navigate to http://localhost:4502/content/wknd-spa-react/us/en.model.json

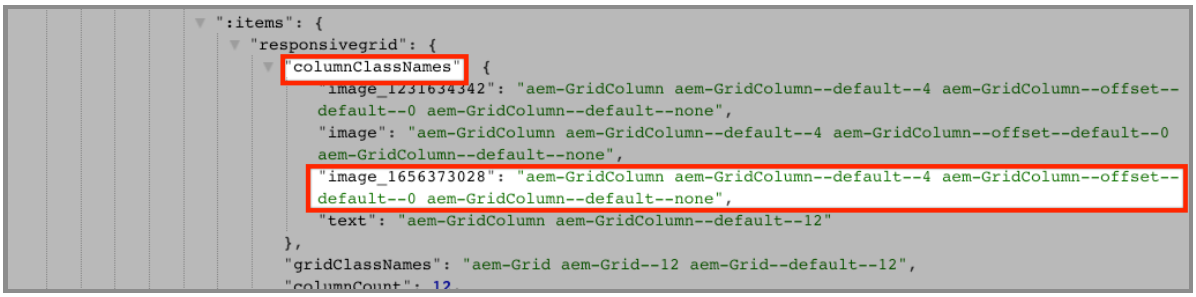
   The **Layout Container** component has a `sling:resourceType` of `wcm/foundation/components/responsivegrid` and is recognized by the SPA Editor using the `:type` property, just like the `Text` and `Image` components.

   The same capabilities of re-sizing a component using Layout Mode are available with the SPA Editor.

2. Return to http://localhost:4502/editor.html/content/wknd-spa-react/us/en/home.html. Add additional **Image** components and try re-sizing them using the **Layout** option:



3. Re-open the JSON model http://localhost:4502/content/wknd-spa-react/us/en.model.json and observe the `columnClassNames` as part of the JSON:

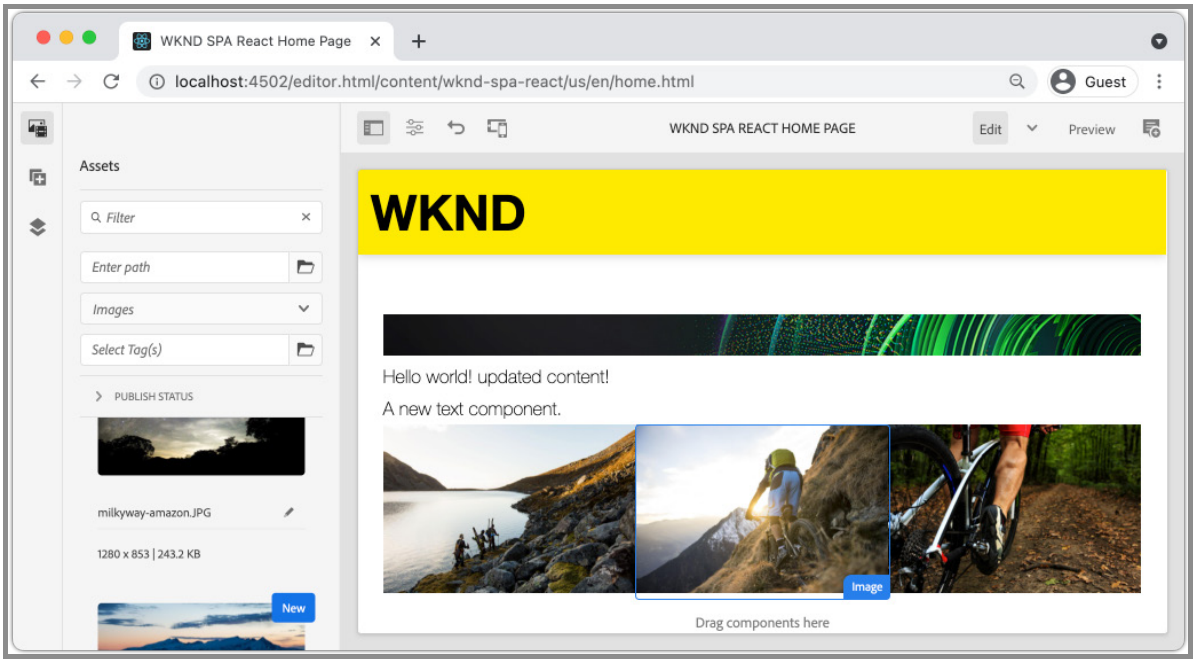


The class name `aem-GridColumn--default--4` indicates the component should be 4 columns wide based on a 12 column grid. More details about the responsive grid can be found here.

4. Return to the IDE and in the `ui.apps` module there is a client-side library defined at `ui.apps/src/main/content/jcr_root/apps/wknd-spa-react/clientlibs/clientlib-grid`. Open the file `less/grid.less`.

   This file determines the breakpoints (`default`, `tablet`, and `phone`) used by the **Layout Container**. This file is intended to be customized per project specifications. Currently the breakpoints are set to `1200px` and `768px`.

5. You should be able to use the responsive capabilities and the updated rich text policies of the `Text` component to author a view like the following:



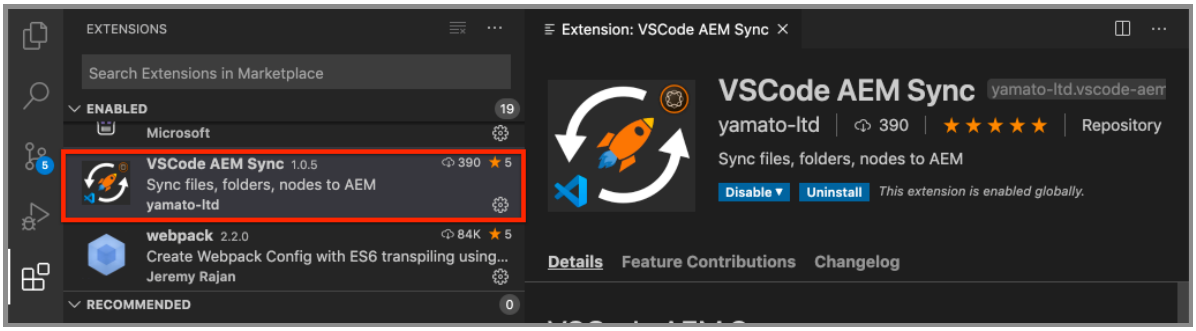
Congratulations, you learned how to map SPA components to AEM Components and you used the React Core Components. You also got a chance to explore the responsive capabilities of the **Layout Container**.

(Bonus) Persist Configurations to Source Control

In many cases, especially at the beginning of an AEM project it is valuable to persist configurations, like templates and related content policies, to source control. This ensures that all developers are working against the same set of content and configurations and can ensure additional consistency between environments. Once a project reaches a certain level of maturity, the practice of managing templates can be turned over to a special group of power users.

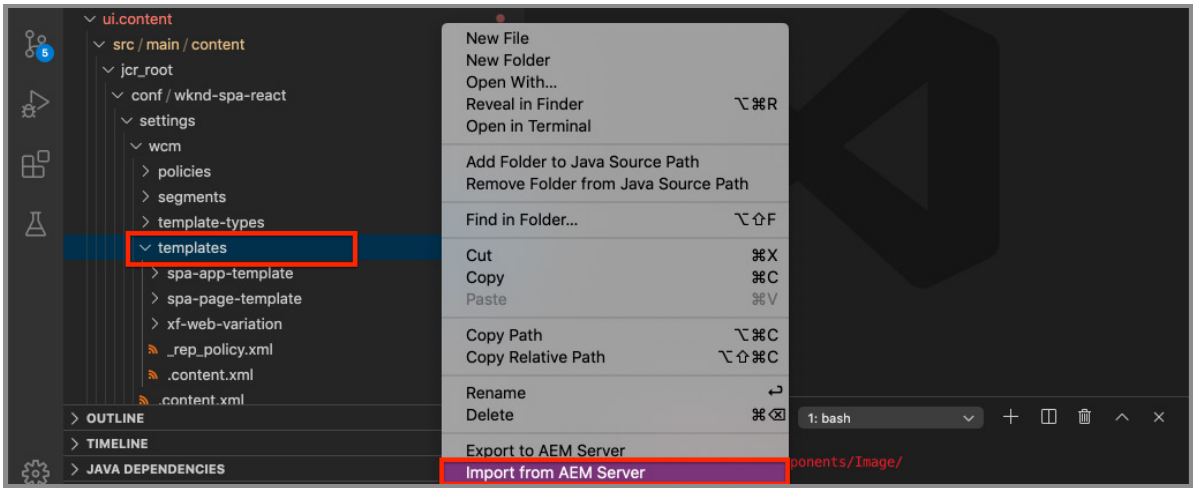
The next few steps will take place using the Visual Studio Code IDE and VSCode AEM Sync but could be doing using any tool and any IDE that you have configured to **pull** or **import** content from a local instance of AEM.

1. In the Visual Studio Code IDE, ensure that you have **VSCode AEM Sync** installed via the Marketplace extension:



2. Expand the **ui.content** module in the Project explorer and navigate to `/conf/wknd-spa-react/settings/wcm/templates`.

3. **Right+Click** the `templates` folder and select **Import from AEM Server**:



4. Repeat the steps to import content but select the **policies** folder located at `/conf/wknd-spa-react/settings/wcm/templates/policies`.

5. Inspect the `filter.xml` file located at `ui.content/src/main/content/META-INF/vault/filter.xml`.

```
<!--ui.content filter.xml-->
<?xml version="1.0" encoding="UTF-8"?>
 <workspaceFilter version="1.0">
     <filter root="/conf/wknd-spa-react" mode="merge"/>
     <filter root="/content/wknd-spa-react" mode="merge"/>
     <filter root="/content/dam/wknd-spa-react" mode="merge"/>
     <filter root="/content/experience-fragments/wknd-spa-react"
mode="merge"/>
 </workspaceFilter>
```

The `filter.xml` file is responsible for identifying the paths of nodes that will be installed with the package. Notice the `mode="merge"` on each of the filters which indicates that existing content will not be modified, only new content is added. Since content authors may be updating these paths, it is important that a code deployment does **not** overwrite content. See the FileVault documentation for more details on working with filter elements.

Compare `ui.content/src/main/content/META-INF/vault/filter.xml` and `ui.apps/src/main/content/META-INF/vault/filter.xml` to understand the different nodes managed by each module.

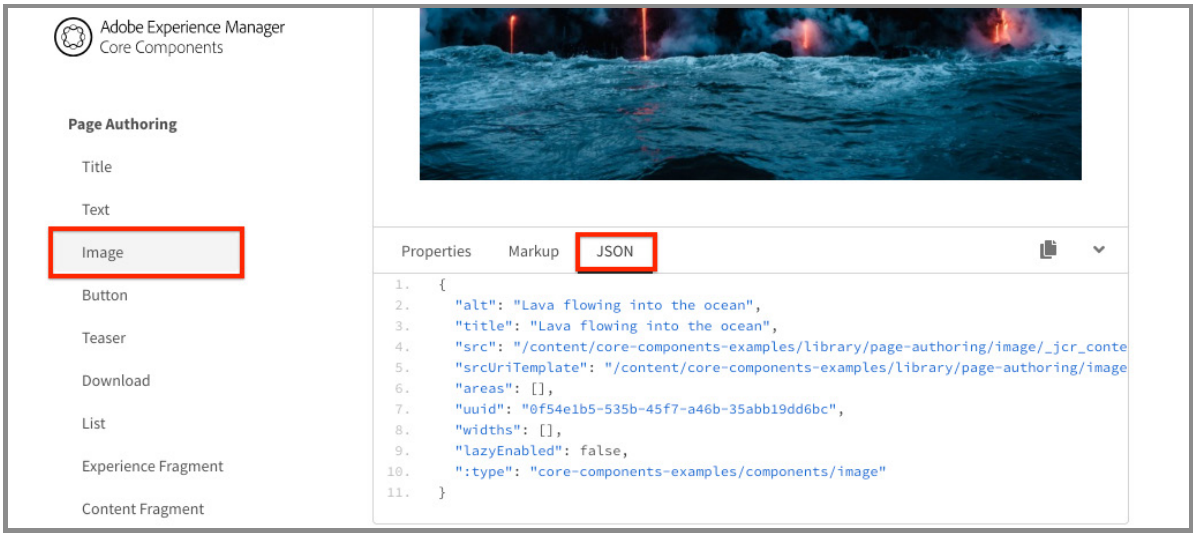
(Bonus) Create Custom Image Component

A SPA Image component has already been provided by the React Core components. However, you can create your own React implementation that maps to the AEM Image component. The `Image` component is another example of a **content** component.

Inspect the JSON

Before jumping into the SPA code, inspect the JSON model provided by AEM.

1. Navigate to the Image examples in the Core Component library.



Properties of `src`, `alt`, and `title` will be used to populate the SPA `Image` component.

NOTE:

There are other Image properties exposed (`lazyEnabled`, `widths`) that allow a developer to create an adaptive and lazy-loading component. The component built in this course will be simple and will **not** use these advanced properties.

Implement the Image Component

1. Next, create a new folder named `Image` under `ui.frontend/src/components`.
2. Beneath the `Image` folder create a new file named `Image.js`.

3. Add the following `import` statements to `Image.js`:

```
import React, {Component} from 'react';
import {MapTo} from '@adobe/aem-react-editable-components';
```

4. Then add the `ImageEditConfig` to determine when to show the placeholder in AEM:

```
export const ImageEditConfig = {

    emptyLabel: 'Image',

    isEmpty: function(props) {
        return !props || !props.src || props.src.trim().length <
1;
    }
};
```

The placeholder will show if the `src` property is not set.

5. Next implement the `Image` class:

```
 export default class Image extends Component {

    get content() {
        return <img     className="Image-src"
                        src={this.props.src}
                        alt={this.props.alt}
                        title={this.props.title ?
this.props.title : this.props.alt} />;
    }

    render() {
        if(ImageEditConfig.isEmpty(this.props)) {
            return null;
        }

        return (
                <div className="Image">
                    {this.content}
                </div>
        );
    }
}
```

The above code will render an `<img>` based on the props `src`, `alt`, and `title` passed in by the JSON model.

6. Add the `MapTo` code to map the React component to the AEM component:

```
MapTo('wknd-spa-react/components/image')(Image, ImageEditConfig);
```

Note the string `wknd-spa-react/components/image` corresponds to the location of the AEM component in `ui.apps` at: `ui.apps/src/main/content/jcr_root/apps/wknd-spa-react/components/image`.

7. Create a new file named `Image.css` in the same directory and add the following:

```
.Image-src {
    margin: 1rem 0;
    width: 100%;
    border: 0;
}
```

8. In `Image.js` add a reference to the file at the top beneath the `import` statements:

```
import React, {Component} from 'react';
import {MapTo} from '@adobe/aem-react-editable-components';

require('./Image.css');
```

9. Open the file `ui.frontend/src/components/import-components.js` and add a reference to the new `Image` component:

```
import './Page/Page';
import './Text/Text';
import './Container/Container';
import './ExperienceFragment/ExperienceFragment';
import './Image/Image'; //add reference to Image component
```

10. In `import-components.js` comment out the React Core Component Image:

```
//MapTo('wknd-spa-react/components/image')(ImageV2, {isEmpty:
ImageV2IsEmptyFn});
```

    This will ensure that our custom Image component is used instead.

11. From the root of the project deploy the SPA code to AEM using Maven:

```
$ cd aem-guides-wknd-spa.react
$ mvn clean install -PautoInstallSinglePackage
```

12. Inspect the SPA in AEM. Any Image components on the page should continue to work. Inspect the rendered output and you should see the markup for our custom Image component instead of the React Core Component.

    Custom Image component markup

    ```
    <div class="Image">
        <img class="Image-src" src="../../../../../../content/image-src.jpg">
    </div>
    ```

    React Core Component Image markup

    ```
    <div class="cmp-image cq-dd-image">
        <img src="../../../../../../content/image-src.jpg" class="cmp-image__image">
    </div>
    ```

    This is a good introduction to extending and implementing your own components.

Add Navigation and Routing

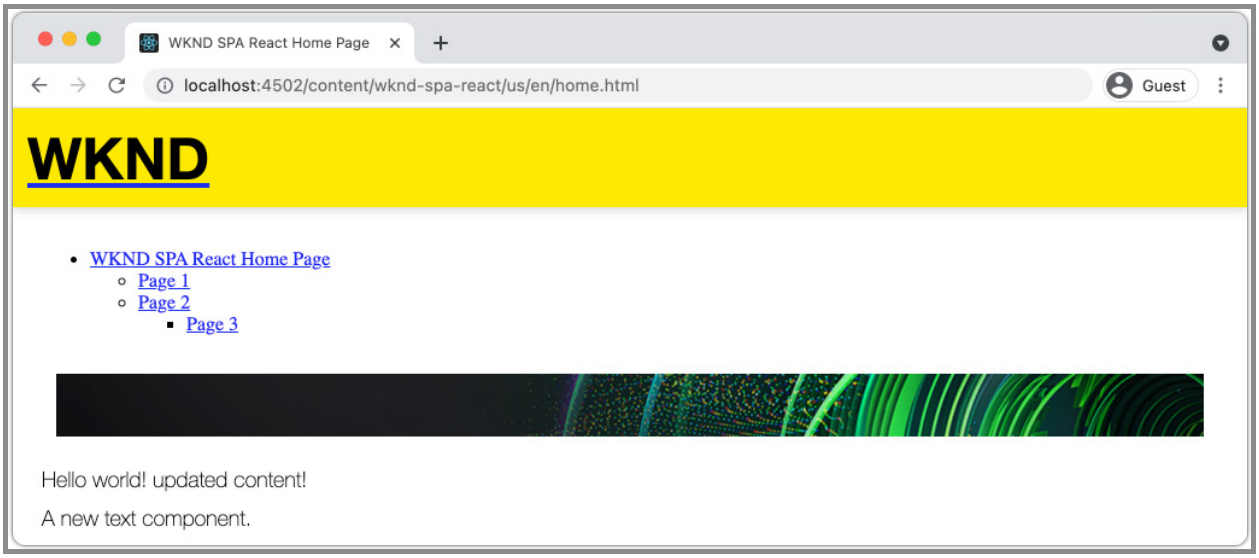
Learn how multiple views in the SPA can be supported by mapping to AEM Pages with the SPA Editor SDK. Dynamic navigation is implemented using React Router and React Core Components.

Objectives

1. Understand the SPA model routing options available when using the SPA Editor.
2. Learn to use React Router to navigate between different views of the SPA.
3. Use AEM React Core Components to implement a dynamic navigation that is driven by the AEM page hierarchy.

What You Will Build

This chapter will add navigation to a SPA in AEM. The navigation menu will be driven by the AEM page hierarchy and will make use of the JSON model provided by the Navigation Core Component.



Prerequisites

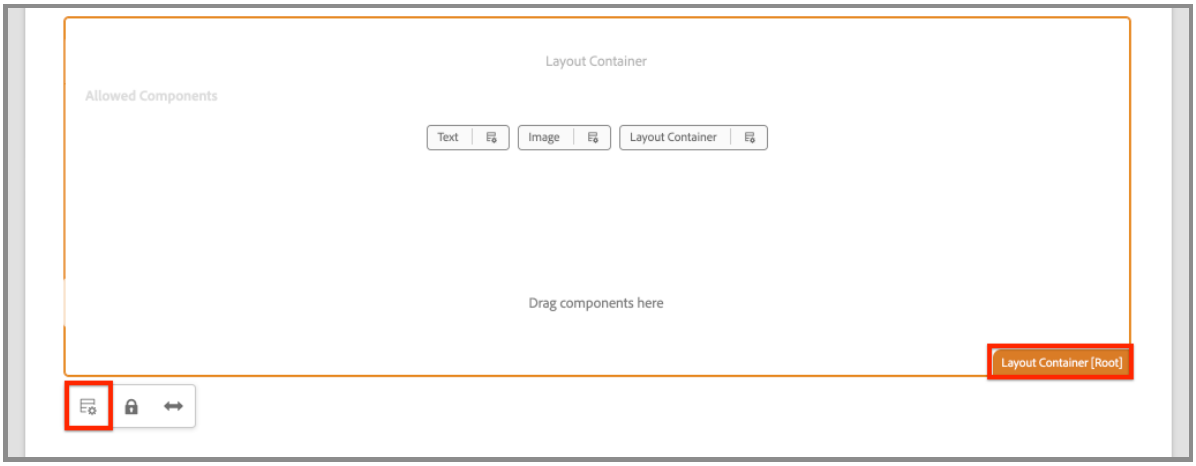
Review the required tooling and instructions for setting up a local development environment. This chapter is a continuation of the Map Components chapter, however to follow along all you need is a SPA-enabled AEM project deployed to a local AEM instance.

Add the Navigation to the Template

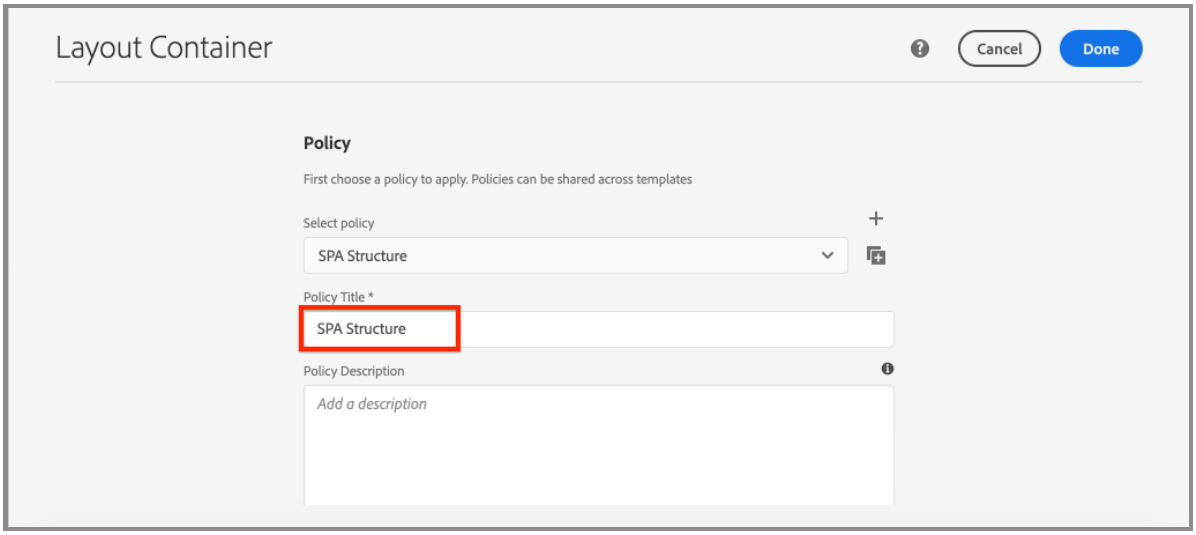
1. Open a browser and login to AEM, http://localhost:4502/. The starting code base should already be deployed.

2. Navigate to the **SPA Page Template**: http://localhost:4502/editor.html/conf/wknd-spa-react/settings/wcm/templates/spa-page-template/structure.html.

3. Select the outer-most **Root Layout Container** and click its **Policy** icon. Be careful **not** to select the **Layout Container** un-locked for authoring.

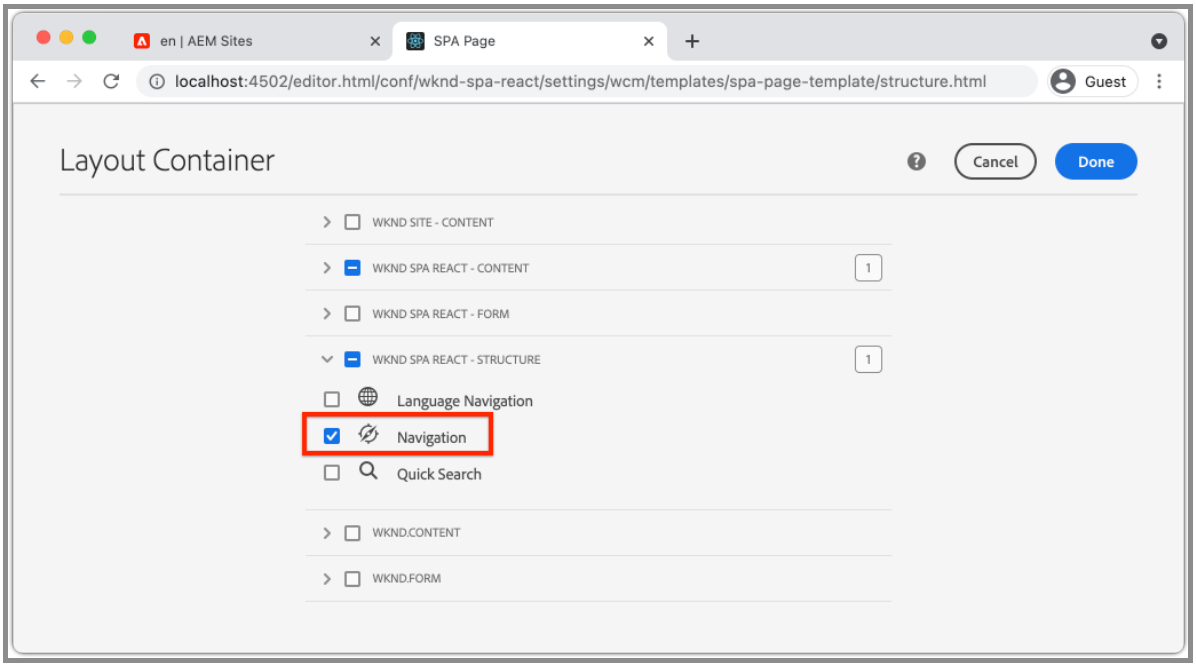


4. Create a new policy named **SPA Structure**:



Under **Allowed Components** > **General** > select the **Layout Container** component.

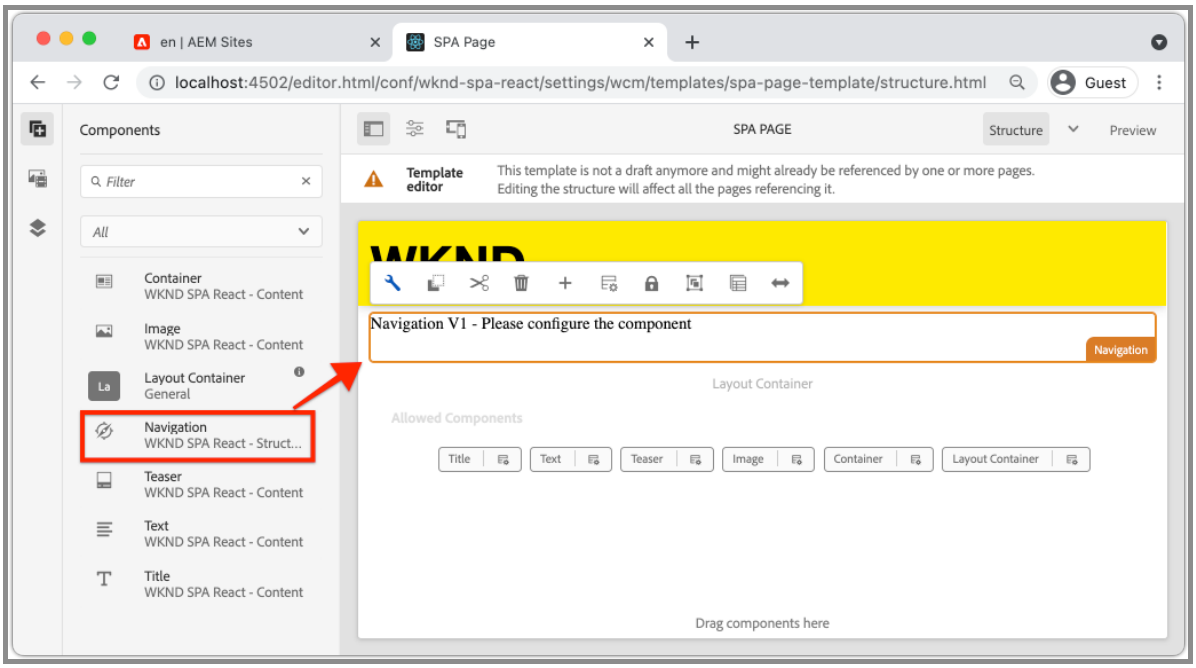
Under **Allowed Components** > **WKND SPA REACT - STRUCTURE** > select the **Header** component:



Under **Allowed Components** > **WKND SPA REACT - Content** > select the **Image** and **Text** components. You should have 4 total components selected.

Click **Done** to save the changes.

5. Refresh the page, and add the **Navigation** component above the un-locked **Layout Container**:

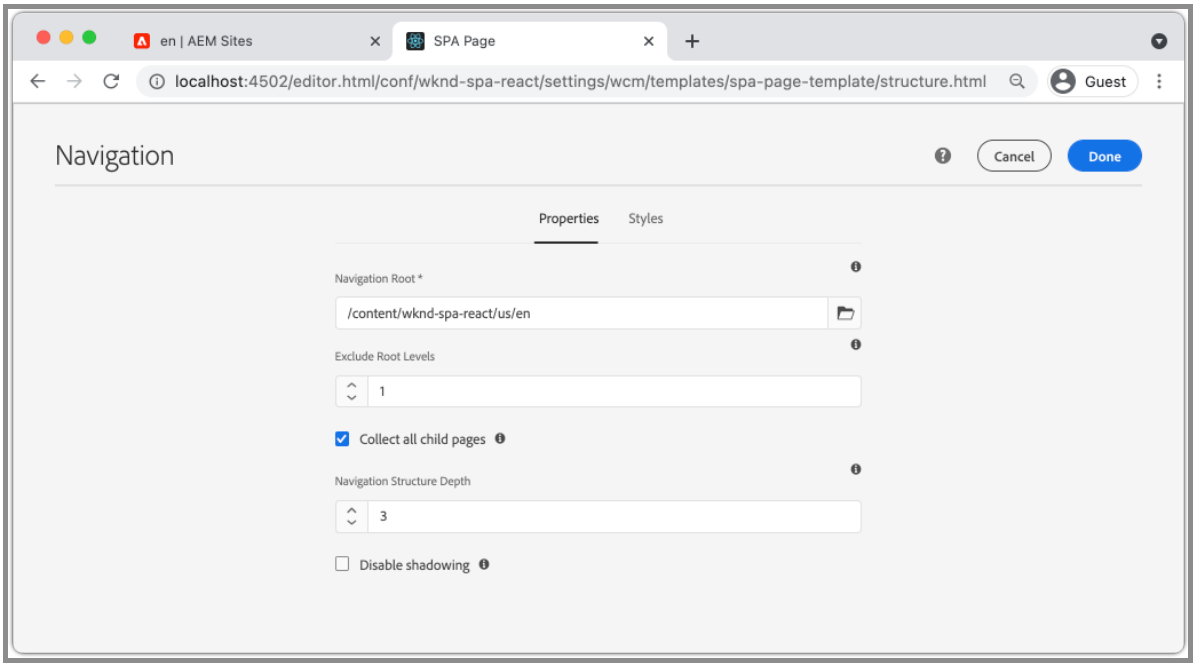


6. Select the **Navigation** component and click its **Policy** icon to edit the policy.

7. Create a new policy with a **Policy Title** of **SPA Navigation**.

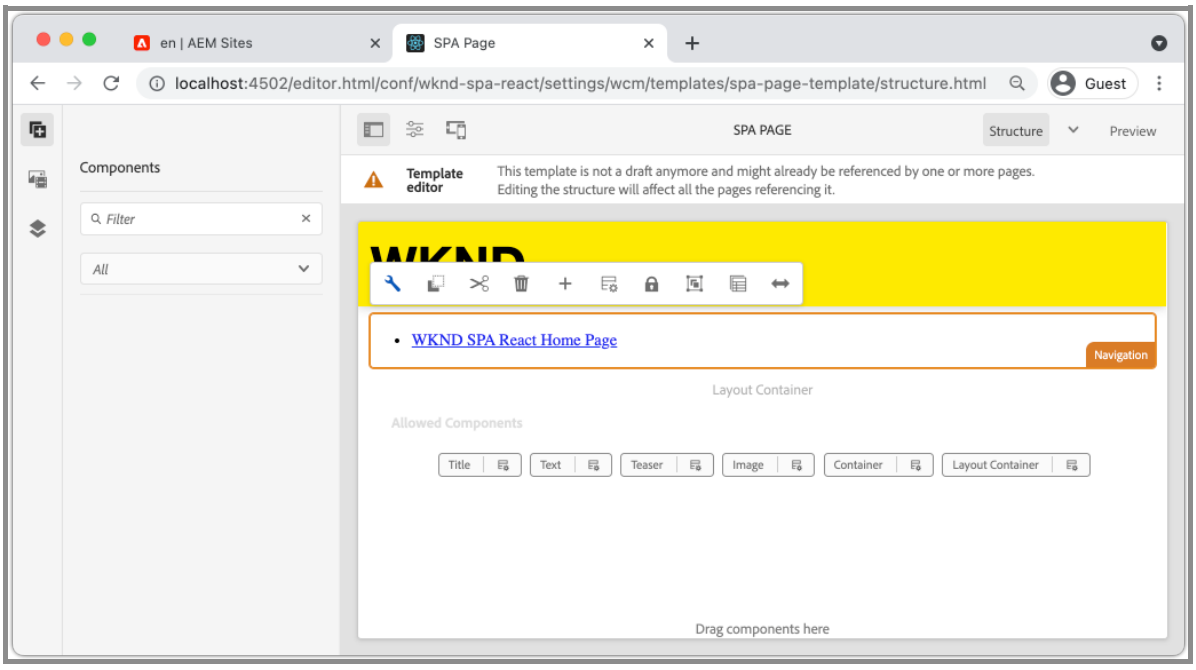
   Under the **Properties**:

   - Set the **Navigation Root** to `/content/wknd-spa-react/us/en`.
   - Set the **Exclude Root Levels** to **1**.
   - Uncheck **Collect all child pages**.
   - Set the **Navigation Structure Depth** to **3**.



   This will collect the navigation 2 levels deep beneath `/content/wknd-spa-react/us/en`.

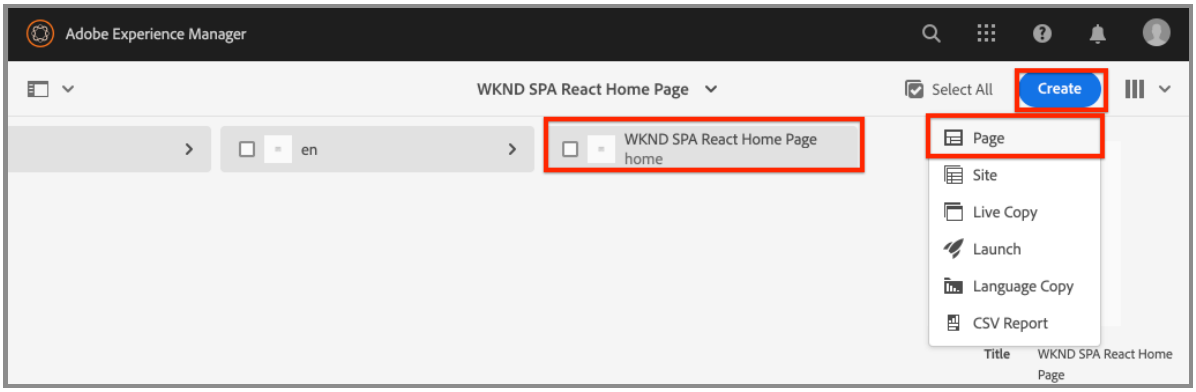
8. After saving your changes you should see the populated `Navigation` as part of the template:



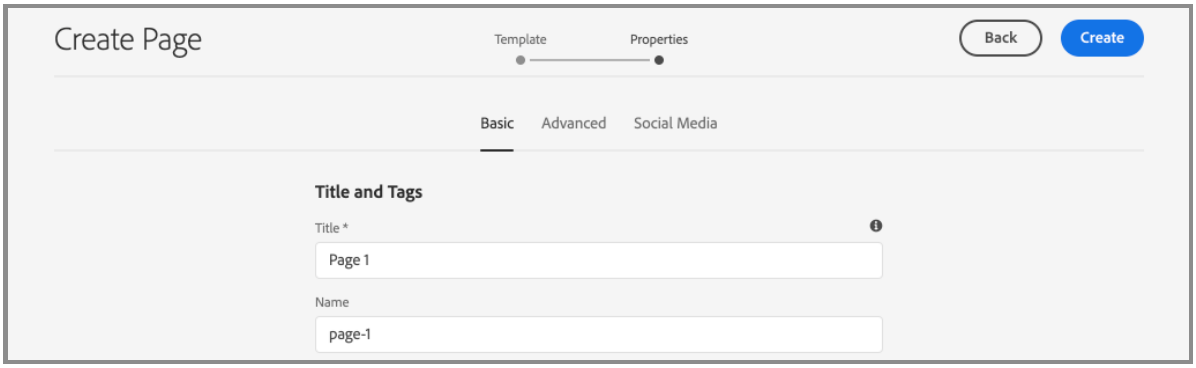
Create Child Pages

Next, create additional pages in AEM that will serve as the different views in the SPA. We will also inspect the hierarchical structure of the JSON model provided by AEM.

1. Navigate to the **Sites** console: http://localhost:4502/sites.html/content/wknd-spa-react/us/en/home. Select the **WKND SPA React Home Page** and click **Create** > **Page**:

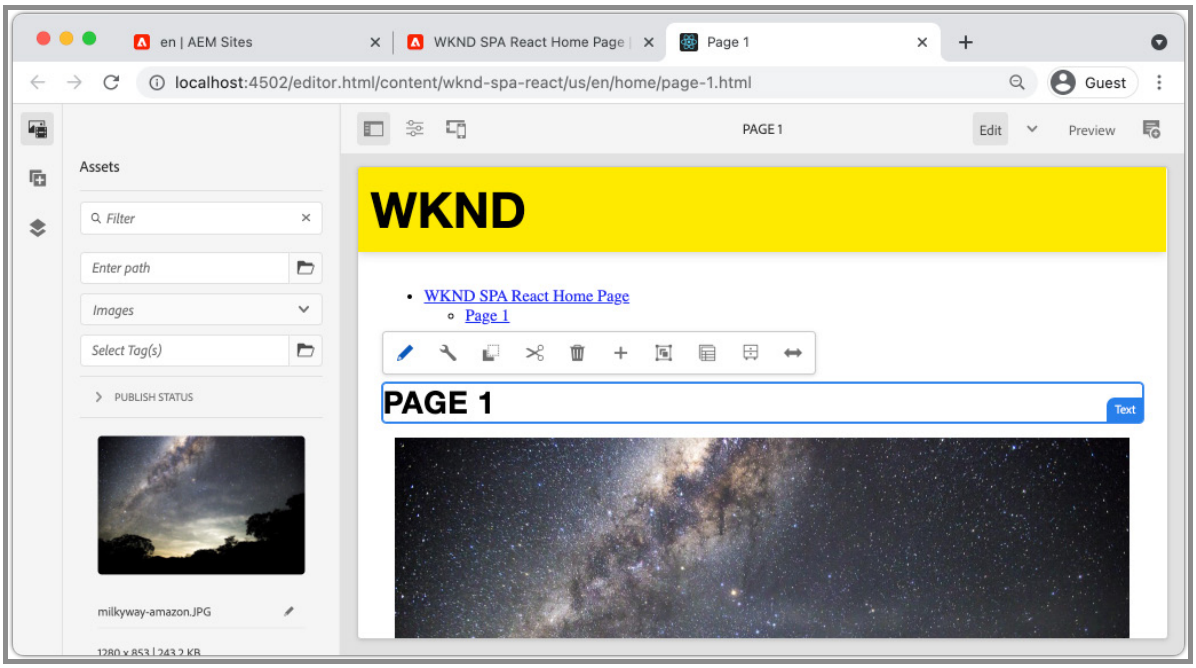


2. Under **Template** select **SPA Page**. Under **Properties** enter **Page 1** for the **Title** and **page-1** as the name.



Click **Create** and in the dialog pop-up, click **Open** to open the page in the AEM SPA Editor.

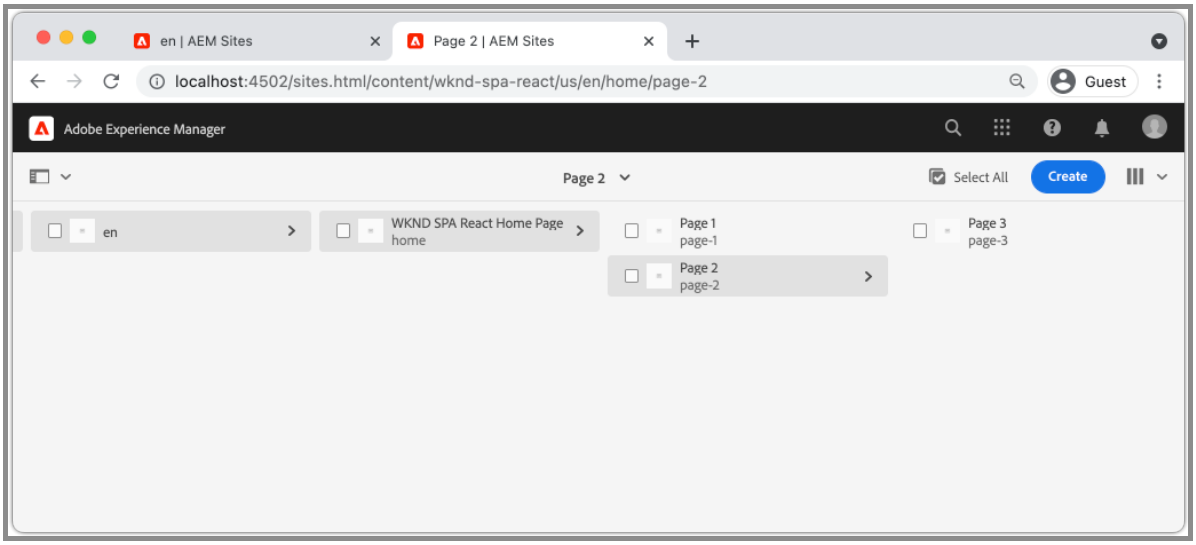
3. Add a new **Text** component to the main **Layout Container**. Edit the component and enter the text: **Page 1** using the RTE and the **H2** element.



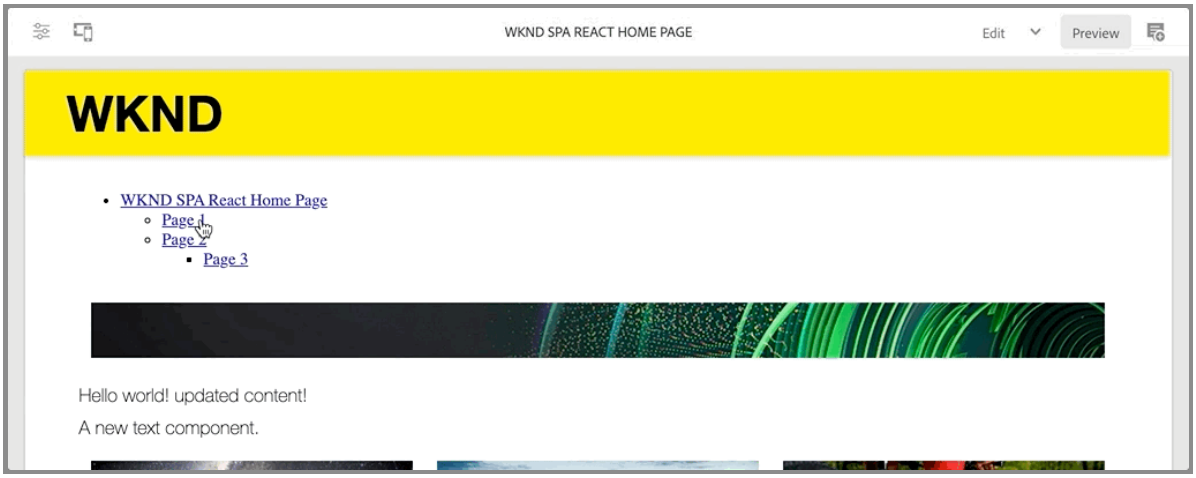
Feel free to add additional content, like an image.

4. Return to the AEM Sites console and repeat the above steps, creating a second page named **Page 2** as a sibling of **Page 1**.

5. Lastly create a third page, **Page 3** but as a **child** of **Page 2**. Once completed the site hierarchy should look like the following:

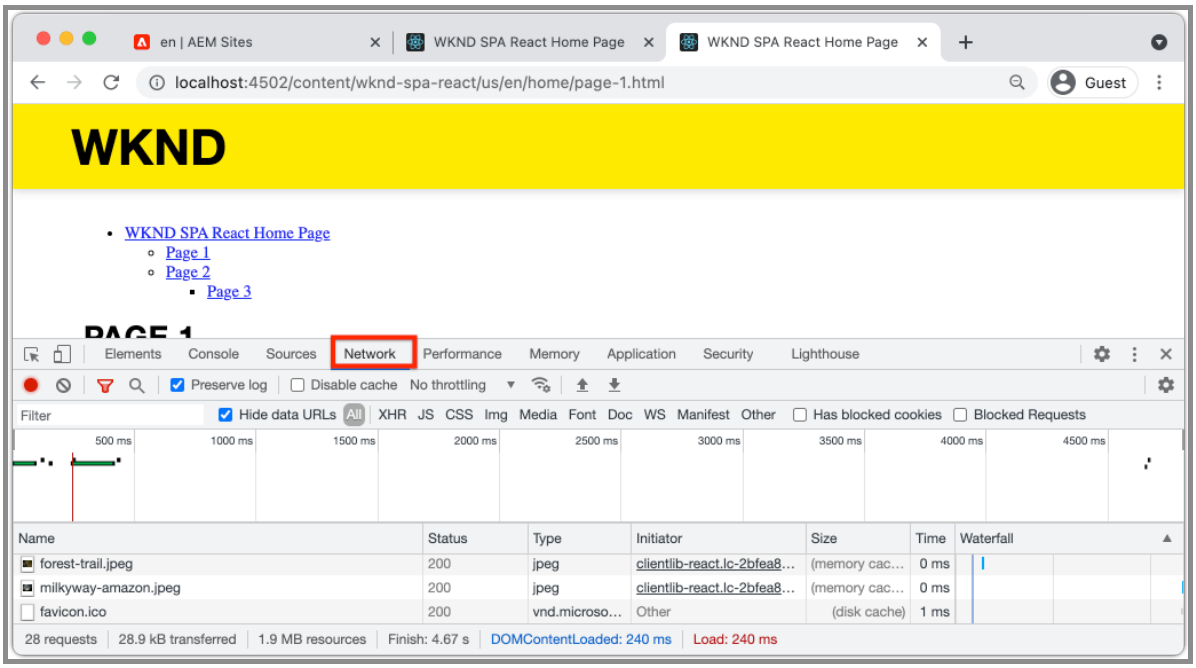


6. The Navigation component can now be used to navigate to different areas of the SPA.



7. Open the page outside of the AEM Editor: http://localhost:4502/content/wknd-spa-react/us/en/home.html. Use the **Navigation** component to navigate to different views of the app.

8. Use your browser's developer tools to inspect the network requests, as you navigate. Screenshots below are captured from Google Chrome browser.



Observe that after the initial page load, subsequent navigation does not cause a full page refresh and that network traffic is minimized when returning to previously visited pages.

Hierarchy Page JSON Model

Next, inspect the JSON Model that drives the multi-view experience of the SPA.

1. In a new tab, open the JSON model API provided by AEM: http://localhost:4502/content/wknd-spa-react/us/en.model.json. It may be helpful to use a browser extension to format the JSON.

   This JSON content is requested when the SPA is first loaded. The outer structure looks like the following:

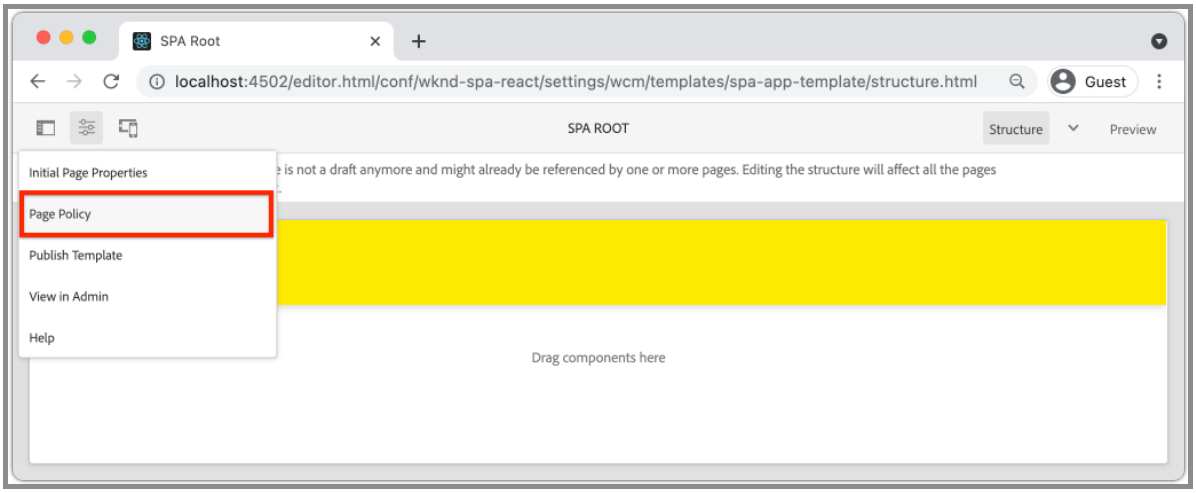
   ```
   {
   "language": "en",
   "title": "en",
   "templateName": "spa-app-template",
   "designPath": "/libs/settings/wcm/designs/default",
   "cssClassNames": "spa page basicpage",
   ":type": "wknd-spa-react/components/spa",
   ":items": {},
   ":itemsOrder": [],
   ":hierarchyType": "page",
   ":path": "/content/wknd-spa-react/us/en",
   ":children": {
      "/content/wknd-spa-react/us/en/home": {},
      "/content/wknd-spa-react/us/en/home/page-1": {},
      "/content/wknd-spa-react/us/en/home/page-2": {},
      "/content/wknd-spa-react/us/en/home/page-2/page-3": {}
      }
   }
   ```

   Under `:children` you should see an entry for each of the pages created. The content for all of the pages is in this initial JSON request. With the navigation routing, subsequent views of the SPA will be loaded rapidly, since the content is already available client-side.

   It is not wise to load **ALL** of the content of a SPA in the initial JSON request, as this would slow down the initial page load. Next, lets look at how the hierarchy depth of pages are collected.

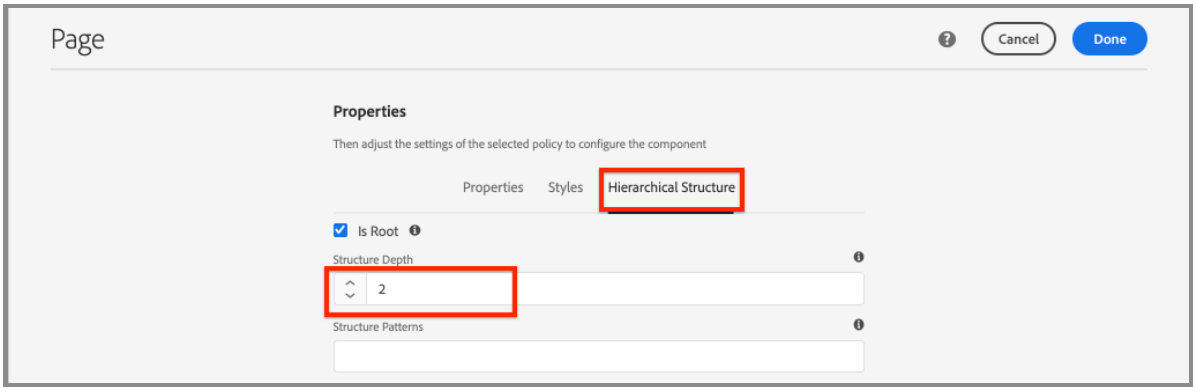
2. Navigate to the **SPA Root** template at: http://localhost:4502/editor.html/conf/wknd-spa-react/settings/wcm/templates/spa-app-template/structure.html.

   Click the **Page properties menu** > **Page Policy**:



3. The **SPA Root** template has an extra **Hierarchical Structure** tab to control the JSON content collected. The **Structure Depth** determines how deep in the site hierarchy to collect child pages beneath the **root**. You can also use the **Structure Patterns** field to filter out additional pages based on a regular expression.

   Update the **Structure Depth** to **2**:



   Click **Done** to save the changes to the policy.

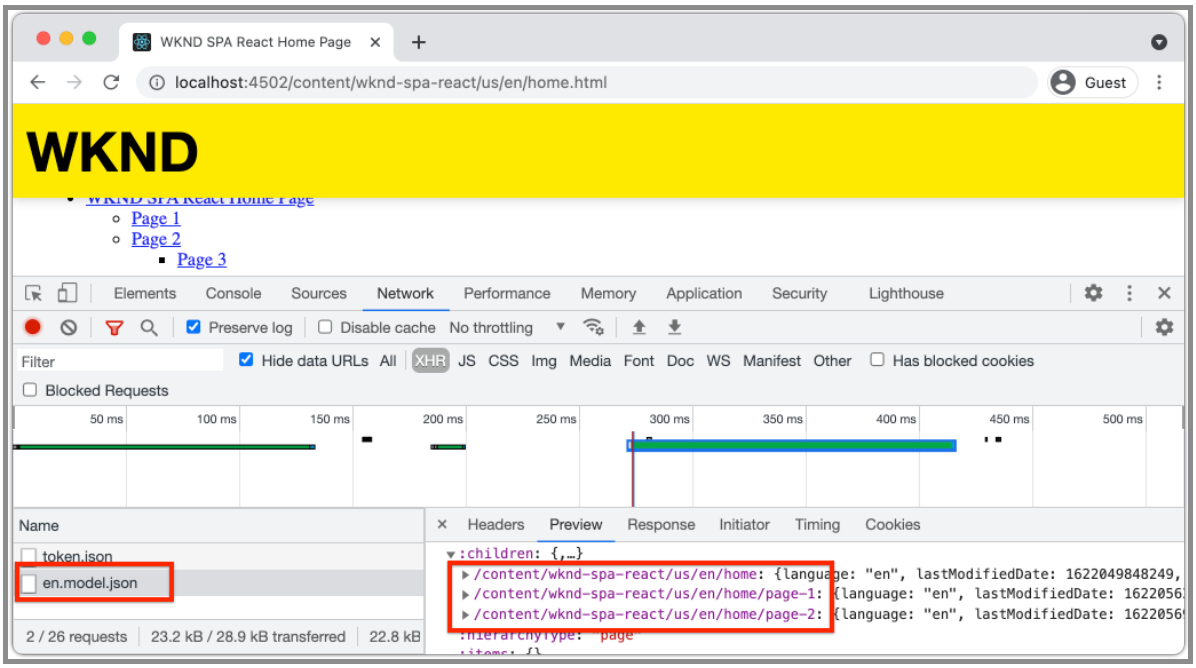
4. Re-open the JSON model http://localhost:4502/content/wknd-spa-react/us/en.model.json.

```
{
"language": "en",
"title": "en",
"templateName": "spa-app-template",
"designPath": "/libs/settings/wcm/designs/default",
"cssClassNames": "spa page basicpage",
":type": "wknd-spa-react/components/spa",
":items": {},
":itemsOrder": [],
":hierarchyType": "page",
":path": "/content/wknd-spa-react/us/en",
":children": {
   "/content/wknd-spa-react/us/en/home": {},
   "/content/wknd-spa-react/us/en/home/page-1": {},
   "/content/wknd-spa-react/us/en/home/page-2": {}
   }
}
```

Notice that the **Page 3** path has been removed: `/content/wknd-spa-react/us/en/home/page-2/page-3` from the initial JSON model. This is because **Page 3** is at a level 3 in the hierarchy and we updated the policy to only include content at a max depth of level 2.

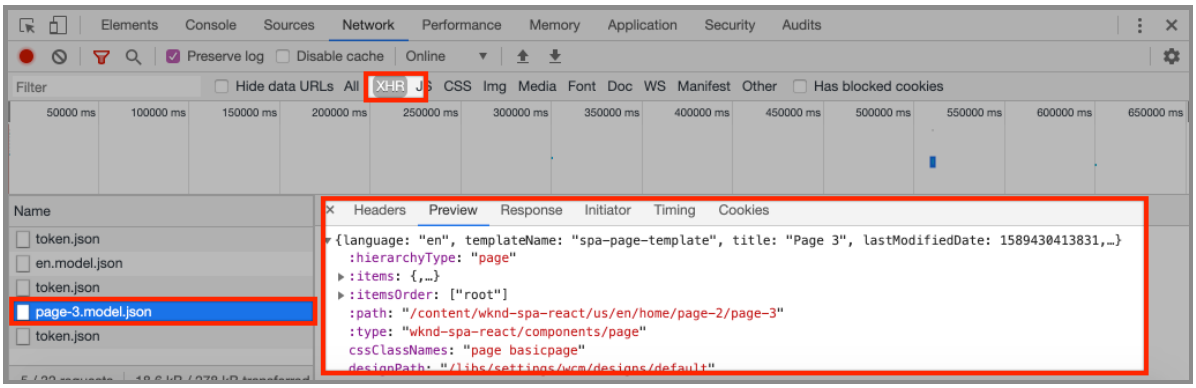
5. Re-open the SPA homepage: http://localhost:4502/content/wknd-spa-react/us/en/home.html and open your browser's developer tools.

   Refresh the page and you should see the XHR request to `/content/wknd-spa-react/us/en.model.json`, which is the SPA Root. Notice that only three child pages are included based on the hierarchy depth configuration to the SPA Root template made earlier in the course. This does not include **Page 3**.



6. With the developer tools open, use the `Navigation` component to navigate directly to **Page 3**:

   Observe that a new XHR request is made to: `/content/wknd-spa-react/us/en/home/page-2/page-3.model.json`



   The AEM Model Manager understands that the **Page 3** JSON content is not available and automatically triggers the additional XHR request.

7. Experiment with deep links by navigating directly to: http://localhost:4502/content/wknd-spa-react/us/en/home/page-2.html. Also observe that the browser's back button continues to work.

Inspect React Routing

The navigation and routing is implemented with React Router. React Router is a collection of navigation components for React applications. AEM React Core Components uses features of React Router to implement the **Navigation** component used in the previous steps.

Next, inspect how React Router is integrated with the SPA and experiment using React Router's Link component.

1. In the IDE open the file `index.js` at `ui.frontend/src/index.js`.

```
/* index.js */
import { Router } from 'react-router-dom';
...
...
 ModelManager.initialize().then(pageModel => {
    const history = createBrowserHistory();
    render(
    <Router history={history}>
        <App
        history={history}
        cqChildren={pageModel[Constants.CHILDREN_PROP]}
        cqItems={pageModel[Constants.ITEMS_PROP]}
        cqItemsOrder={pageModel[Constants.ITEMS_ORDER_PROP]}
        cqPath={pageModel[Constants.PATH_PROP]}
        locationPathname={window.location.pathname}
        />
    </Router>,
    document.getElementById('spa-root')
    );
});
```

Notice that the `App` is wrapped in the `Router` component from React Router. The `ModelManager`, provided by the AEM SPA Editor JS SDK, adds the dynamic routes to AEM Pages based on the JSON model API.

2. Open the file `Page.js` at `ui.frontend/src/components/Page/Page.js`

```
class AppPage extends Page {
  get containerProps() {
    let attrs = super.containerProps;
    attrs.className =
      (attrs.className || '') + ' page ' +
(this.props.cssClassNames || '');
    return attrs;
  }
}

export default MapTo('wknd-spa-react/components/page')(
  withComponentMappingContext(withRoute(AppPage))
);
```

The `Page` SPA component uses the `MapTo` function to map **Pages** in AEM to a corresponding SPA component. The `withRoute` utility helps to dynamically route the SPA to the appropriate AEM Child page based on the `cqPath` property.

3. Open the `Header.js` component at `ui.frontend/src/components/Header/Header.js`.

4. Update the `Header` to wrap the `<h1>` tag in a Link to the homepage:

```
  //Header.js
  import React, {Component} from 'react';
+ import {Link} from 'react-router-dom';
  require('./Header.css');

export default class Header extends Component {

    render() {
        return (
            <header className="Header">
            <div className="Header-container">
+              <Link to="/content/wknd-spa-
react/us/en/home.html">
                    <h1>WKND</h1>
+              </Link>
            </div>
            </header>
        );
    }
```

Instead of using a default `<a>` anchor tag we use `<Link>` provided by React Router. As long as the `to=` points to a valid route, the SPA will switch to that route and **not** perform a full page refresh. Here we simply hard-code the link to the home page to illustrate the use of `Link`.

5. Update the test at `App.test.js` at `ui.frontend/src/App.test.js`.

```
+ import { BrowserRouter as Router } from 'react-router-dom';
  import App from './App';

  it('renders without crashing', () => {
    const div = document.createElement('div');
-   ReactDOM.render(<App />, div);
+   ReactDOM.render(<Router><App /></Router>, div);
  });
```

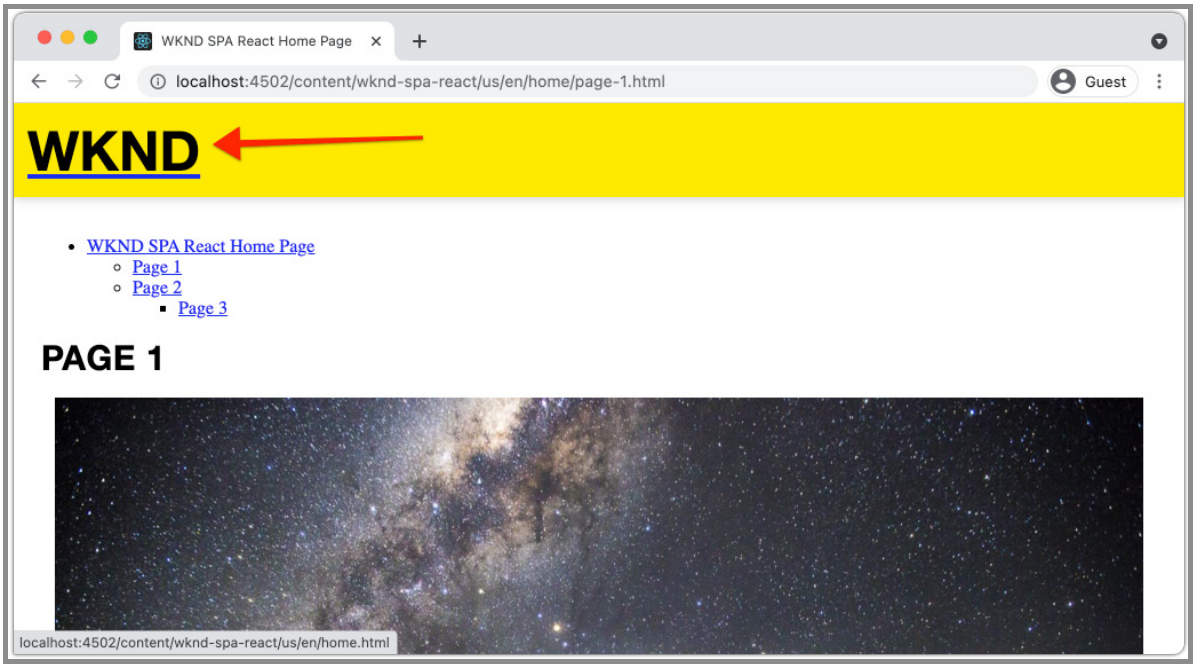
Since we are using features of React Router within a static component referenced in `App.js` we need to update the unit test to account for it.

6. Open a terminal, navigate to the root of the project, and deploy the project to AEM using your Maven skills:

```
$ cd aem-guides-wknd-spa.react
$ mvn clean install -PautoInstallSinglePackage
```

7. Navigate to one of the pages in the SPA in AEM: http://localhost:4502/content/wknd-spa-react/us/en/home/page-1.html

   Instead of using the `Navigation` component to navigate, use the link in the `Header`.



   Observe that a full page refresh is **not** triggered and that the SPA routing is working.

8. Optionally, experiment with the `Header.js` file using a standard `<a>` anchor tag:

```
<a href="/content/wknd-spa-react/us/en/home.html">
    <h1>WKND</h1>
</a>
```

   This can help illustrate the difference between SPA routing and regular web page links.

Congratulations, you learned how multiple views in the SPA can be supported by mapping to AEM Pages with the SPA Editor SDK. Dynamic navigation has been implemented using React Router and added to the `Header` component.

Create a Custom Weather Component

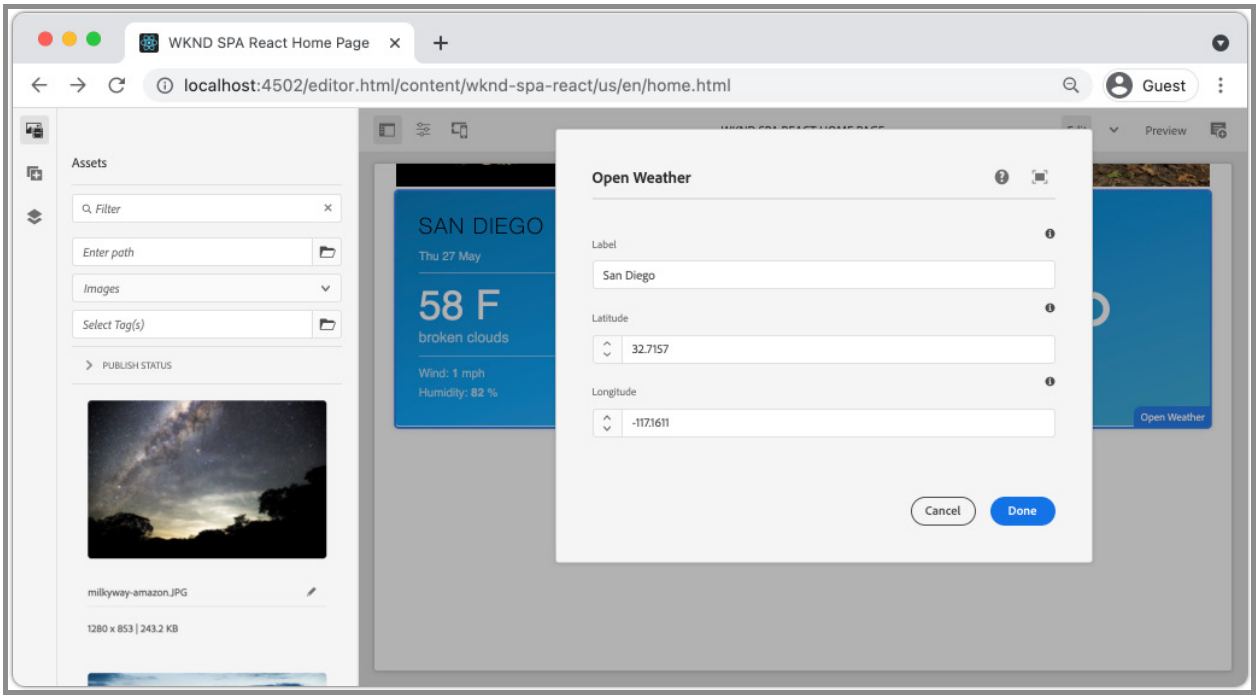
Learn how to create a custom weather component to be used with the AEM SPA Editor. Learn how to develop author dialogs and Sling Models to extend the JSON model to populate a custom component. The Open Weather API and React Open Weather component are used.

Objectives

1. Understand the role of Sling Models in manipulating the JSON model API provided by AEM.
2. Understand how to create new AEM component dialogs.
3. Learn to create a **custom** AEM Component that will be compatible with the SPA editor framework.

What You Will Build

A simple weather component will be built. This component will be able to be added to the SPA by content authors. Using an AEM dialog, authors can set the location for the weather to be displayed. The implementation of this component illustrates the steps needed to create a net-new AEM component that is compatible with the AEM SPA Editor framework.



Prerequisites

Review the required tooling and instructions for setting up a local development environment. This chapter is a continuation of the Navigation and Routing chapter, however to follow along all you need is a SPA-enabled AEM project deployed to a local AEM instance.

Open Weather API Key

Your instructor will provide you with a key to use for this project.

You can also get your own API key from Open Weather. Sign up is free for a limited amount of API calls.

Define the AEM Component

An AEM component is defined as a node and properties. In the project these nodes and properties are represented as XML files in the `ui.apps` module. Next, create the AEM component in the `ui.apps` module.

NOTE:

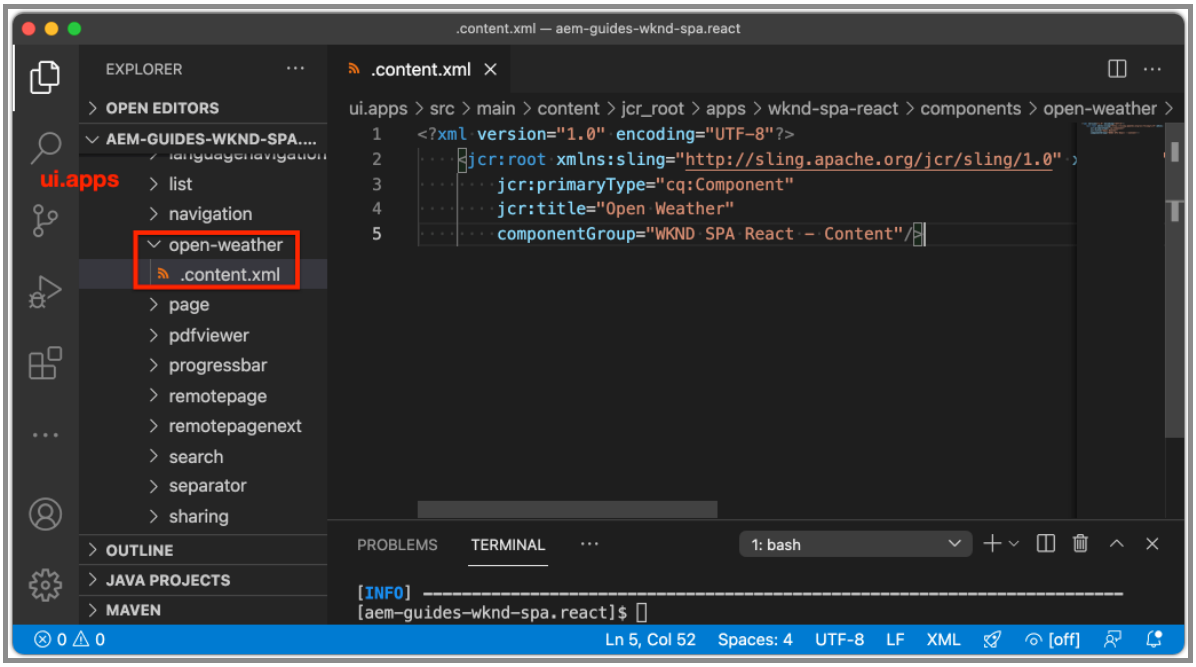
A quick refresher on the basics of AEM components may be helpful.

---

1. In the IDE of your choice open the `ui.apps` folder.
2. Navigate to `ui.apps/src/main/content/jcr_root/apps/wknd-spa-react/components` and create a new folder named `open-weather`.

3. Create a new file named `.content.xml` beneath the `open-weather` folder. Populate the `open-weather/.content.xml` with the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:sling="http://sling.apache.org/jcr/sling/1.0"
xmlns:cq="http://www.day.com/jcr/cq/1.0"
xmlns:jcr="http://www.jcp.org/jcr/1.0"
    jcr:primaryType="cq:Component"
    jcr:title="Open Weather"
    componentGroup="WKND SPA React - Content"/>
```



`jcr:primaryType="cq:Component"` - identifies that this node will be an AEM component.

`jcr:title` is the value that will be displayed to Content Authors and the `componentGroup` determines the grouping of components in the authoring UI.

4. Beneath the `custom-component` folder, create another folder named `_cq_dialog`.

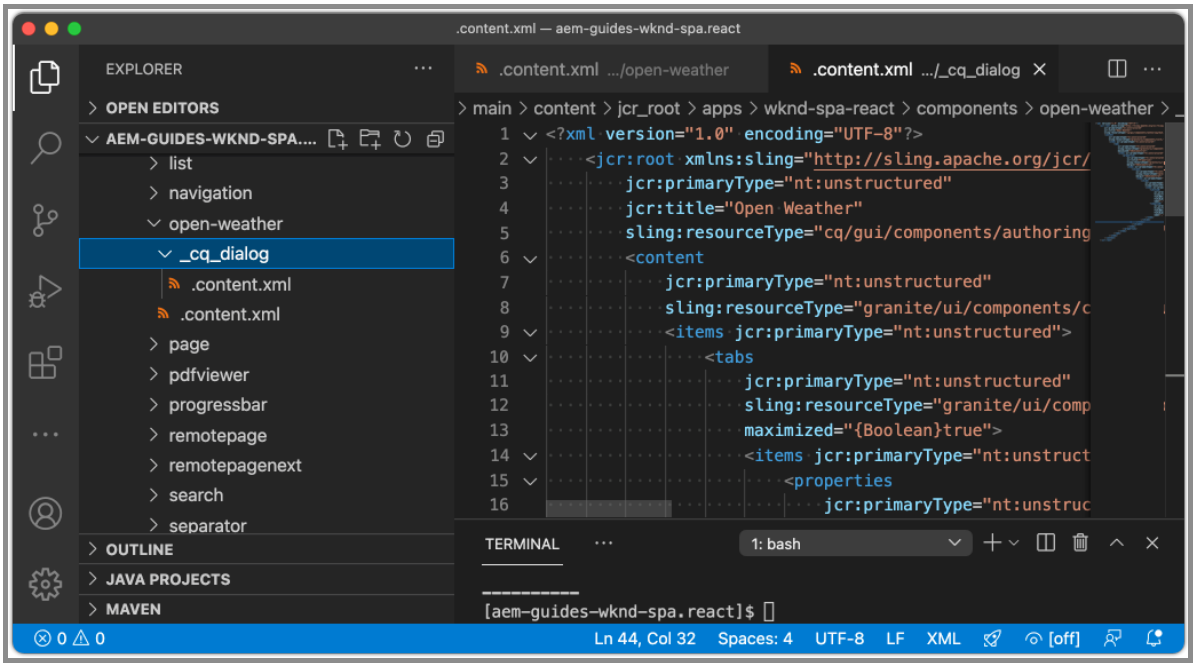
5. Beneath the `_cq_dialog` folder create a new file named `.content.xml` and populate it with the following:

```
<?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="http://www.day.com/jcr/cq/1.0"
xmlns:jcr="http://www.jcp.org/jcr/1.0"
xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
```

```
    jcr:primaryType="nt:unstructured"
    jcr:title="Open Weather"
    sling:resourceType="cq/gui/components/authoring/dialog">
    <content
        jcr:primaryType="nt:unstructured"

 sling:resourceType="granite/ui/components/coral/foundation/conta
iner">
        <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">
                    <properties
                        jcr:primaryType="nt:unstructured"
                        jcr:title="Properties"

 sling:resourceType="granite/ui/components/coral/foundation/conta
iner"
                        margin="{Boolean}true">
                        <items jcr:primaryType="nt:unstructured">
                            <columns
                                jcr:primaryType="nt:unstructured"

 sling:resourceType="granite/ui/components/coral/foundation/fixed
columns"
                                margin="{Boolean}true">
                                <items
jcr:primaryType="nt:unstructured">
                                    <column

 jcr:primaryType="nt:unstructured"

 sling:resourceType="granite/ui/components/coral/foundation/conta
iner">
                                        <items
jcr:primaryType="nt:unstructured">
```

```
                                        <label

 jcr:primaryType="nt:unstructured"

 sling:resourceType="granite/ui/components/coral/foundation/form/
textfield"

 fieldDescription="The label to display for the component"

 fieldLabel="Label"
                                            name="./label"/>
                                        <lat

 jcr:primaryType="nt:unstructured"

 sling:resourceType="granite/ui/components/coral/foundation/form/
numberfield"

 fieldDescription="The latitude of the location."

 fieldLabel="Latitude"
                                            step="any"
                                            name="./lat" />
                                        <lon

 jcr:primaryType="nt:unstructured"

 sling:resourceType="granite/ui/components/coral/foundation/form/
numberfield"

 fieldDescription="The longitude of the location."

 fieldLabel="Longitude"
                                            step="any"
                                            name="./lon"/>
                                    </items>
                                </column>
                            </items>
                        </columns>
```

```
                        </items>
                    </properties>
                </items>
            </tabs>
        </items>
    </content>
</jcr:root>
```



The above XML file generates a very simple dialog for the `Weather Component`. The critical part of the file is the inner `<label>`, `<lat>` and `<lon>` nodes. This dialog will contain two `numberfield`s and a `textfield` that will allow a user to configure the the weather to be displayed.

A Sling Model will be created next to expose the value of the `label`, `lat` and `long` properties via the JSON model.

NOTE:

You can view a lot more examples of dialogs by viewing the Core Component definitions. You can also view additional form fields, like `select`, `textarea`, `pathfield`, available beneath `/libs/granite/ui/components/coral/foundation/form` in CRXDE-Lite.

With a traditional AEM component, an HTL script is typically required. Since the SPA will render the component, no HTL script is needed.

Create the Sling Model

Sling Models are annotation driven Java "POJO's" (Plain Old Java Objects) that facilitate the mapping of data from the JCR to Java variables. Sling Models typically function to encapsulate complex server-side business logic for AEM Components.

In the context of the SPA Editor, Sling Models expose a component's content through the JSON model through a feature using the Sling Model Exporter.

1. In the IDE of your choice open the `core` module at `aem-guides-wknd-spa.react/core`.
2. Create a file named at `OpenWeatherModel.java` at `core/src/main/java/com/adobe/aem/guides/wkndspa/react/core/models`.
3. Populate `OpenWeather.java` with the following:

```
package com.adobe.aem.guides.wkndspa.react.core.models;

import com.adobe.cq.export.json.ComponentExporter;

// Sling Models intended to be used with SPA Editor must extend
ComponentExporter interface
public interface OpenWeatherModel extends ComponentExporter {

    public String getLabel();

    public double getLat();

    public double getLon();

}
```

   This is the Java interface for our component. In order for our Sling Model to be compatible with the SPA Editor framework it must extend the `ComponentExporter` class.

4. Create a folder named `impl` beneath `core/src/main/java/com/adobe/aem/guides/wkndspa/react/core/models`.
5. Create a file named `OpenWeatherModelImpl.java` beneath `impl` and populate with the following:

```
package com.adobe.aem.guides.wkndspa.react.core.models.impl;
```

```
import org.apache.sling.models.annotations.*;
import
org.apache.sling.models.annotations.injectorspecific.ValueMapValu
e;
import com.adobe.cq.export.json.ComponentExporter;
import com.adobe.cq.export.json.ExporterConstants;
import org.apache.commons.lang3.StringUtils;
import org.apache.sling.api.SlingHttpServletRequest;
import
com.adobe.aem.guides.wkndspa.react.core.models.OpenWeatherModel;

// Sling Model annotation
@Model(
    adaptables = SlingHttpServletRequest.class,
    adapters = { OpenWeatherModel.class, ComponentExporter.class
},
    resourceType = OpenWeatherModelImpl.RESOURCE_TYPE,
    defaultInjectionStrategy = DefaultInjectionStrategy.OPTIONAL
    )
@Exporter( //Exporter annotation that serializes the modoel as
JSON
    name = ExporterConstants.SLING_MODEL_EXPORTER_NAME,
    extensions = ExporterConstants.SLING_MODEL_EXTENSION
    )
public class OpenWeatherModelImpl implements OpenWeatherModel {

    @ValueMapValue
    private String label; //maps variable to jcr property named
"label" persisted by Dialog

    @ValueMapValue
    private double lat; //maps variable to jcr property named
"lat"

    @ValueMapValue
    private double lon; //maps variable to jcr property named
"lon"
```

```
    // points to AEM component definition in ui.apps
    static final String RESOURCE_TYPE = "wknd-spa-
react/components/open-weather";

    // public getter method to expose value of private variable
`label`
    // adds additional logic to default the label to "(Default)"
if not set.
    @Override
    public String getLabel() {
        return StringUtils.isNotBlank(label) ? label : "
(Default)";
    }

    // public getter method to expose value of private variable
`lat`
    @Override
    public double getLat() {
        return lat;
    }

    // public getter method to expose value of private variable
`lon`
    @Override
    public double getLon() {
        return lon;
    }

    // method required by `ComponentExporter` interface
    // exposes a JSON property named `:type` with a value of
`wknd-spa-react/components/open-weather`
    // required to map the JSON export to the SPA component props
via the `MapTo`
    @Override
    public String getExportedType() {
        return OpenWeatherModelImpl.RESOURCE_TYPE;
    }
}
```

The the static variable `RESOURCE_TYPE` must point to the path in `ui.apps` of the component. The `getExportedType()` is used to map the JSON properties to the SPA component via `MapTo`. `@ValueMapValue` is an annotation that reads the jcr property saved by the dialog.

Update the SPA

Next, update the React code to include the React Open Weather component and have it map to the AEM component created in the previous steps.

1. Install the React Open Weather component as an **npm** dependency:

```
$ cd aem-guides-wknd-spa.react/ui.frontend
$ npm i react-open-weather
```

2. Create a new folder named `OpenWeather` at `ui.frontend/src/components/OpenWeather`.

3. Add a file named `OpenWeather.js` and populate it with the following:

```
import React from 'react';
import {MapTo} from '@adobe/aem-react-editable-components';
import ReactWeather, { useOpenWeather } from 'react-open-
weather';

// Open weather API Key
// For simplicity it is hard coded in the file, ideally this is
extracted in to an environment variable
const API_KEY = 'YOUR_API_KEY';

// Logic to render placeholder or component
const OpenWeatherEditConfig = {

    emptyLabel: 'Weather',
    isEmpty: function(props) {
        return !props || !props.lat || !props.lon ||
!props.label;
    }
};

// Wrapper function that includes react-open-weather component
function ReactWeatherWrapper(props) {
    const { data, isLoading, errorMessage } = useOpenWeather({
        key: API_KEY,
        lat: props.lat, // passed in from AEM JSON
        lon: props.lon, // passed in from AEM JSON
```

```
        lang: 'en',
        unit: 'imperial', // values are (metric, standard,
imperial)
    });

    return (
        <div className="cmp-open-weather">
            <ReactWeather
                isLoading={isLoading}
                errorMessage={errorMessage}
                data={data}
                lang="en"
                locationLabel={props.label} // passed in from AEM
JSON
                unitsLabels={{ temperature: 'F', windSpeed: 'mph'
}}
                showForecast={false}
              />
        </div>
    );
}

export default function OpenWeather(props) {

        // render nothing if component not configured
        if(OpenWeatherEditConfig.isEmpty(props)) {
            return null;
        }

        // render ReactWeather component if component configured
        // pass props to ReactWeatherWrapper. These props include
the mapped properties from AEM JSON
        return ReactWeatherWrapper(props);

}

// Map OpenWeather to AEM component
MapTo('wknd-spa-react/components/open-weather')(OpenWeather,
OpenWeatherEditConfig);
```

4. Update `import-components.js` at `ui.frontend/src/components/import-components.js` to include the `OpenWeather` component:

```
  // import-component.js
  import './Container/Container';
  import './ExperienceFragment/ExperienceFragment';
+ import './OpenWeather/OpenWeather';
```

5. Deploy all of the updates to a local AEM environment from the root of the project directory, using your Maven skills:

```
$ cd aem-guides-wknd-spa.react
$ mvn clean install -PautoInstallSinglePackage
```

Update the Template Policy

Next, navigate to AEM to verify the updates and allow the `OpenWeather` component to be added to the SPA.

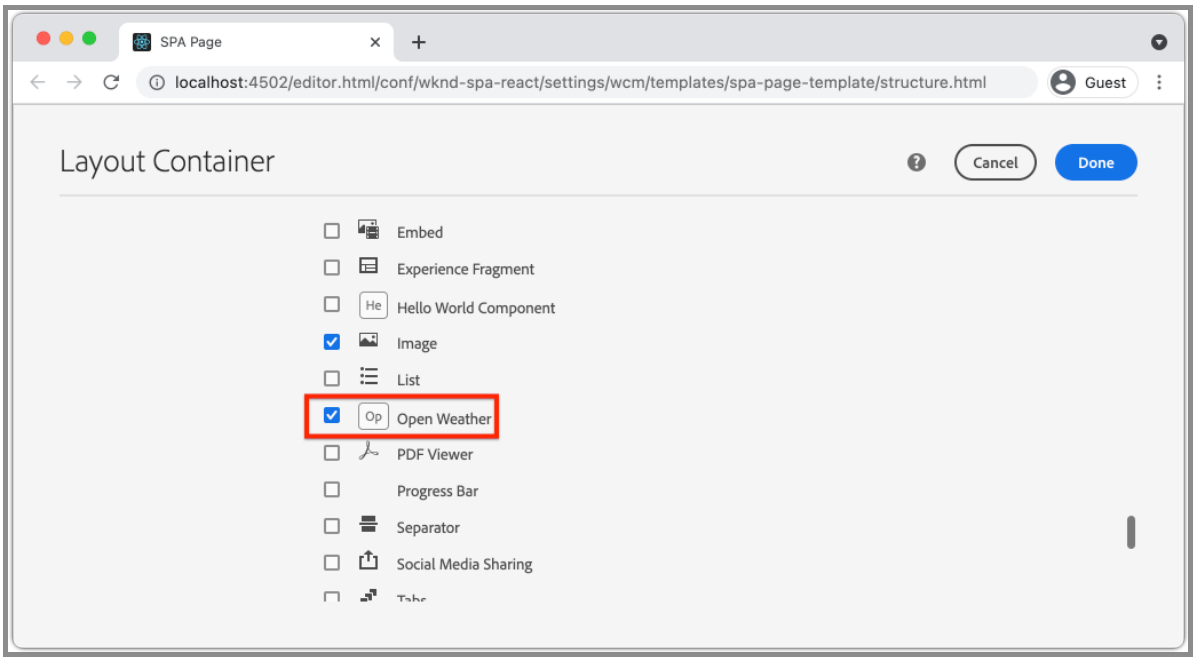
1. Verify the registration of the new Sling Model by navigating to http://localhost:4502/system/console/status-slingmodels.

   ```
   com.adobe.aem.guides.wkndspa.react.core.models.impl.OpenWeatherMo
   delImpl - wknd-spa-react/components/open-weather

   com.adobe.aem.guides.wkndspa.react.core.models.impl.OpenWeatherMo
   delImpl exports 'wknd-spa-react/components/open-weather' with
   selector 'model' and extension '[Ljava.lang.String;@2fd80fc5'
   with exporter 'jackson'
   ```

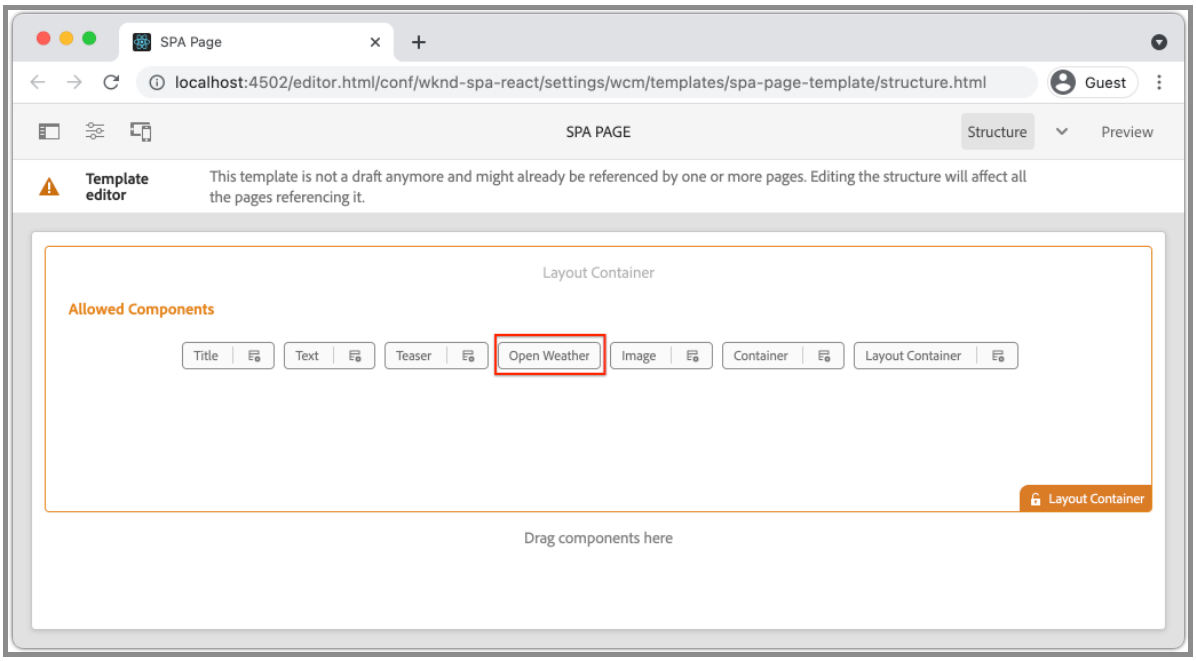
   You should see the above two lines that indicate the `OpenWeatherModelImpl` is associated with the `wknd-spa-react/components/open-weather` component and that it is registered via the Sling Model Exporter.

2. Navigate to the SPA Page Template at http://localhost:4502/editor.html/conf/wknd-spa-react/settings/wcm/templates/spa-page-template/structure.html.

3. Update the Layout Container's policy to add the new `Open Weather` as an allowed component:



Save the changes to the policy, and observe the `Open Weather` as an allowed component:

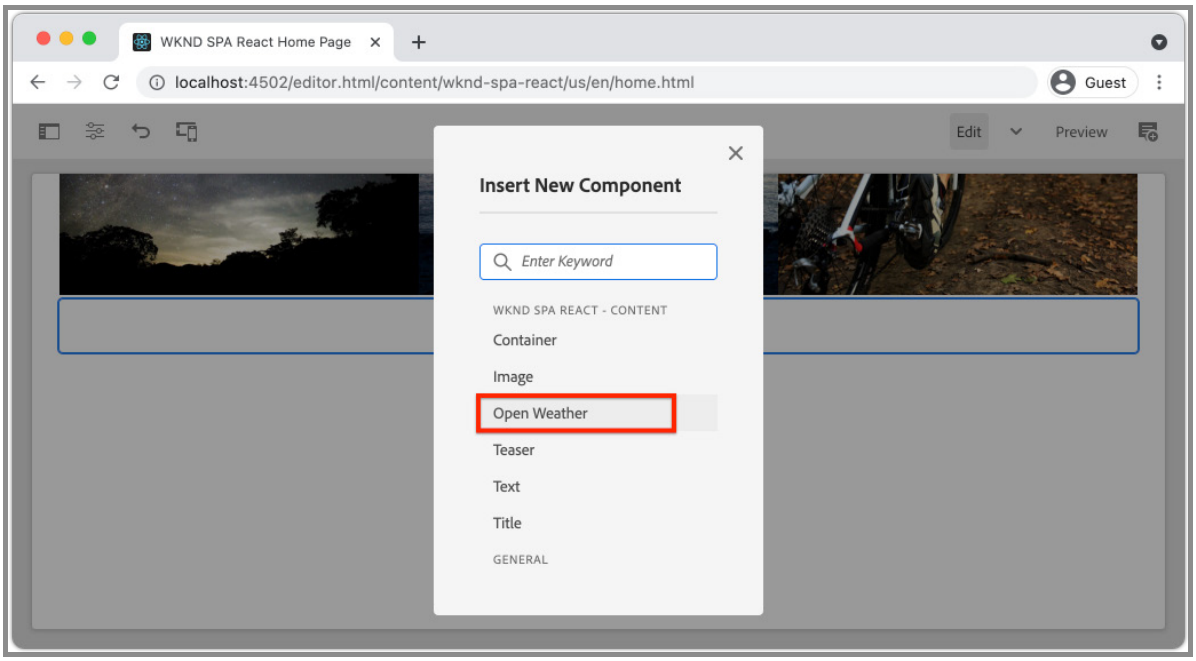


Author the Open Weather Component

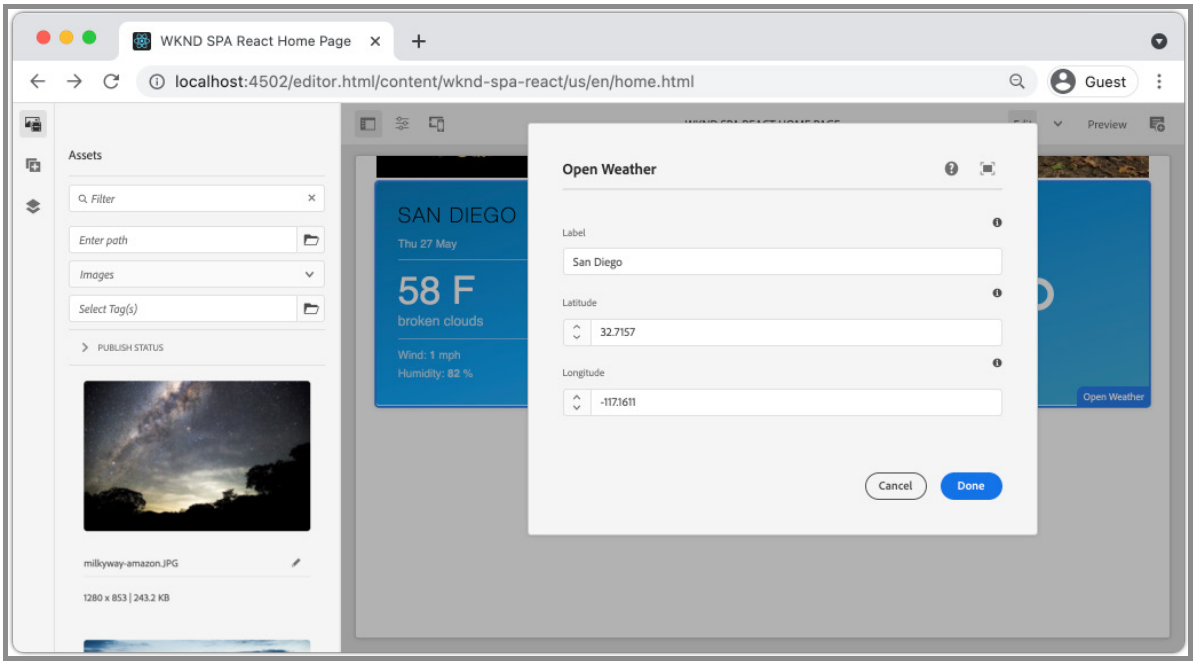
Next, author the `Open Weather` component using the AEM SPA Editor.

1. Navigate to http://localhost:4502/editor.html/content/wknd-spa-react/us/en/home.html.

2. In `Edit` mode, add the `Open Weather` to the `Layout Container`:

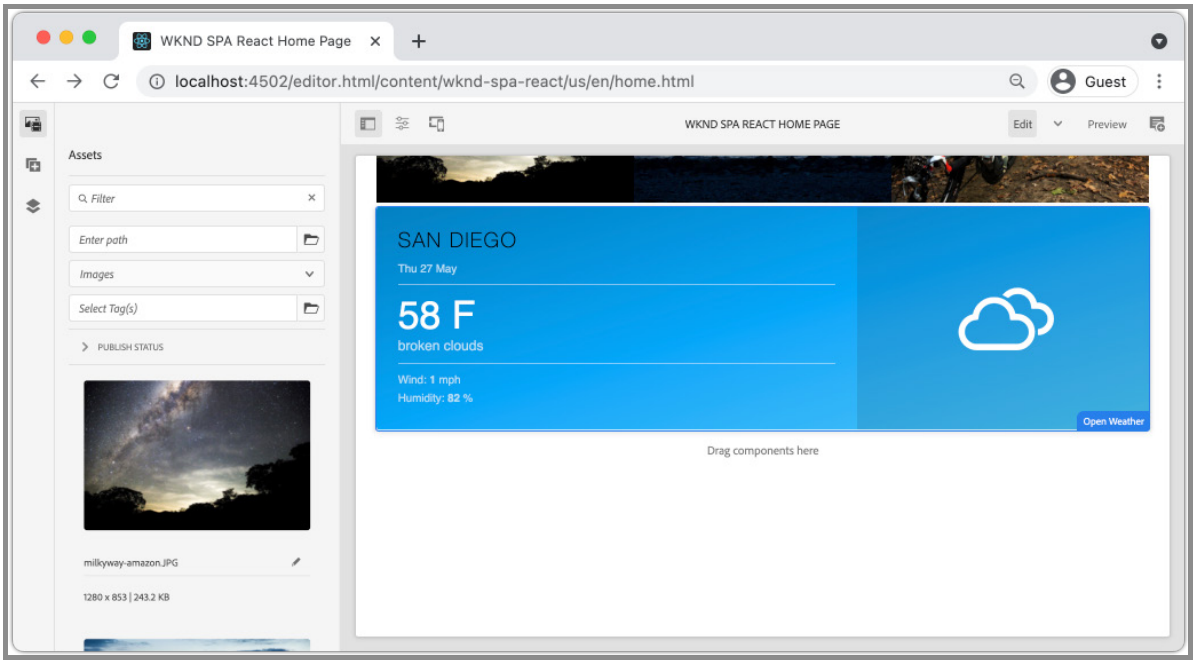


3. Open the component's dialog and enter a **Label**, **Latitude**, and **Longitude**. For example **San Diego**, **32.7157**, and **-117.1611**. Western hemisphere and Southern hemisphere numbers are represented as negative numbers with the Open Weather API



This is the dialog that was created based on the XML file earlier in the chapter.

4. Save the changes. Observe that the weather for **San Diego** is now displayed:



5. View the JSON model by navigating to http://localhost:4502/content/wknd-spa-react/us/en.model.json. Search for `wknd-spa-react/components/open-weather`:

```
"open_weather": {
    "label": "San Diego",
    "lat": 32.7157,
    "lon": -117.1611,
    ":type": "wknd-spa-react/components/open-weather"
}
```

   The JSON values are outputted by the Sling Model. These JSON values are passed into the React component as props.

Congratulations, you learned how to create a custom AEM component to be used with the SPA Editor. You also learned how dialogs, JCR properties, and Sling Models interact to output the JSON model.

Extend a Core Component

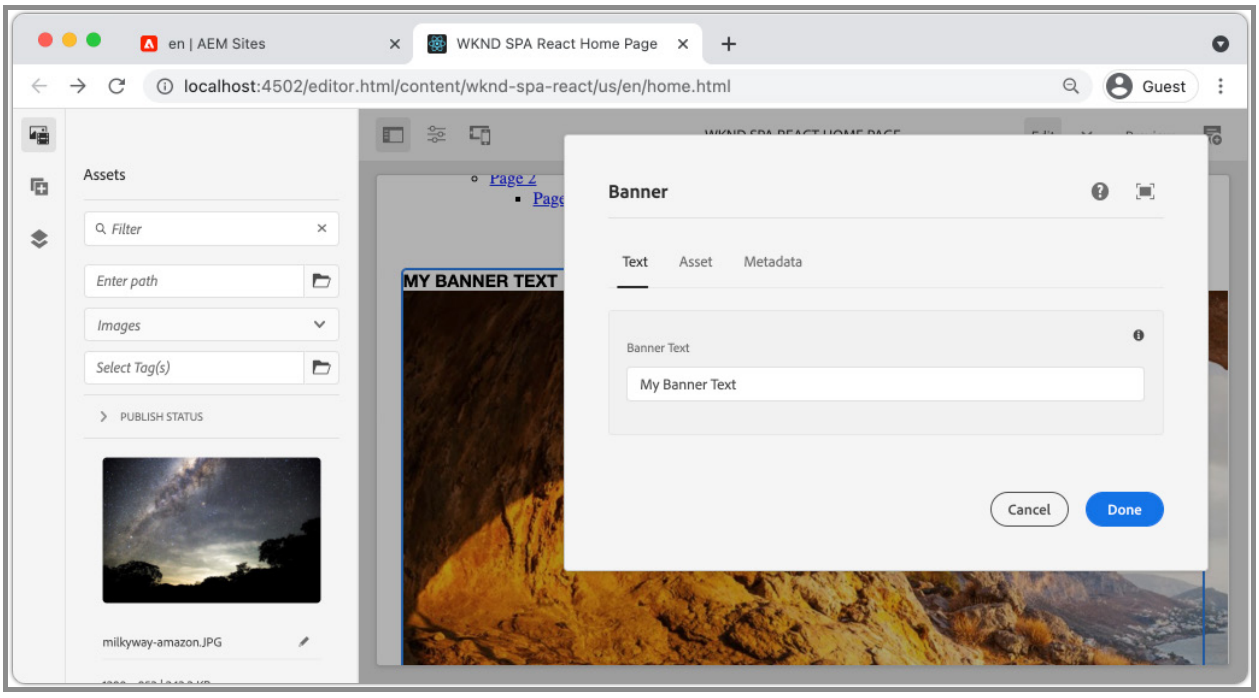
Learn how to extend an existing Core Component to be used with the AEM SPA Editor. Understanding how to extend an existing component is a powerful technique to customize and expand the capabilities of an AEM SPA Editor implementation.

Objectives

1. Extend an existing Core Component with additional properties and content.
2. Understand the basic of Component Inheritance with the use of `sling:resourceSuperType`.
3. Learn how to leverage the Delegation Pattern for Sling Models to re-use existing logic and functionality.

What You Will Build

This chapter illustrates the additional code needed to add an extra property to a standard `Image` component to fulfill the requirements for a new `Banner` component. The `Banner` component contains all of the same properties as the standard `Image` component but includes an additional property for users to populate the **Banner Text**.



Prerequisites

Review the required tooling and instructions for setting up a local development environment. It is assumed at this point in the course users have a solid understanding of the AEM SPA Editor feature.

Inheritance with Sling Resource Super Type

To extend an existing component set a property named `sling:resourceSuperType` on your component's definition. `sling:resourceSuperType` is a property that can be set on an AEM component's definition that points to another component. This explicitly sets the component to inherit all functionality of the component identified as the `sling:resourceSuperType`.

If we want to extend the `Image` component at `wknd-spa-react/components/image` we need to update the code in the `ui.apps` module.

1. Create a new folder beneath the `ui.apps` module for `banner` at `ui.apps/src/main/content/jcr_root/apps/wknd-spa-react/components/banner`.

2. Beneath `banner` create a Component definition (`.content.xml`) like the following:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <jcr:root xmlns:sling="http://sling.apache.org/jcr/sling/1.0"
   xmlns:cq="http://www.day.com/jcr/cq/1.0"
   xmlns:jcr="http://www.jcp.org/jcr/1.0"
       jcr:primaryType="cq:Component"
       jcr:title="Banner"
       sling:resourceSuperType="wknd-spa-react/components/image"
       componentGroup="WKND SPA React - Content"/>
   ```

   This sets `wknd-spa-react/components/banner` to inherit all functionality of `wknd-spa-react/components/image`.

cq:editConfig

The `_cq_editConfig.xml` file dictates the drag and drop behavior in the AEM authoring UI. When extending the Image component it is important that the resource type matches the component itself.

1. In the `ui.apps` module create another file beneath `banner` named `_cq_editConfig.xml`.

2. Populate `_cq_editConfig.xml` with the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:sling="http://sling.apache.org/jcr/sling/1.0"
xmlns:cq="http://www.day.com/jcr/cq/1.0"
xmlns:jcr="http://www.jcp.org/jcr/1.0"
xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
    jcr:primaryType="cq:EditConfig">
    <cq:dropTargets jcr:primaryType="nt:unstructured">
        <image
            jcr:primaryType="cq:DropTargetConfig"
            accept="
[image/gif,image/jpeg,image/png,image/webp,image/tiff,image/svg\\
+xml]"
            groups="[media]"
            propertyName="./fileReference">
            <parameters
                jcr:primaryType="nt:unstructured"
                sling:resourceType="wknd-spa-
react/components/banner"
                imageCrop=""
                imageMap=""
                imageRotate=""/>
        </image>
    </cq:dropTargets>
    <cq:inplaceEditing
        jcr:primaryType="cq:InplaceEditingConfig"
        active="{Boolean}true"
        editorType="image">
        <inplaceEditingConfig jcr:primaryType="nt:unstructured">
            <plugins jcr:primaryType="nt:unstructured">
```

```
                <crop
                    jcr:primaryType="nt:unstructured"
                    supportedMimeTypes="
[image/jpeg,image/png,image/webp,image/tiff]"
                    features="*">
                    <aspectRatios
jcr:primaryType="nt:unstructured">
                        <wideLandscape
                            jcr:primaryType="nt:unstructured"
                            name="Wide Landscape"
                            ratio="0.6180"/>
                        <landscape
                            jcr:primaryType="nt:unstructured"
                            name="Landscape"
                            ratio="0.8284"/>
                        <square
                            jcr:primaryType="nt:unstructured"
                            name="Square"
                            ratio="1"/>
                        <portrait
                            jcr:primaryType="nt:unstructured"
                            name="Portrait"
                            ratio="1.6180"/>
                    </aspectRatios>
                </crop>
                <flip
                    jcr:primaryType="nt:unstructured"
                    supportedMimeTypes="
[image/jpeg,image/png,image/webp,image/tiff]"
                    features="-"/>
                <map
                    jcr:primaryType="nt:unstructured"
                    supportedMimeTypes="
[image/jpeg,image/png,image/webp,image/tiff,image/svg+xml]"
                    features="*"/>
                <rotate
                    jcr:primaryType="nt:unstructured"
                    supportedMimeTypes="
[image/jpeg,image/png,image/webp,image/tiff]"
```

```
                    features="*"/>
                <zoom
                    jcr:primaryType="nt:unstructured"
                    supportedMimeTypes="
[image/jpeg,image/png,image/webp,image/tiff]"
                    features="*"/>
            </plugins>
            <ui jcr:primaryType="nt:unstructured">
                <inline
                    jcr:primaryType="nt:unstructured"
                    toolbar="
[crop#launch,rotate#right,history#undo,history#redo,fullscreen#fu
llscreen,control#close,control#finish]">
                    <replacementToolbars
                        jcr:primaryType="nt:unstructured"
                        crop="
[crop#identifier,crop#unlaunch,crop#confirm]"/>
                </inline>
                <fullscreen jcr:primaryType="nt:unstructured">
                    <toolbar
                        jcr:primaryType="nt:unstructured"
                        left="
[crop#launchwithratio,rotate#right,flip#horizontal,flip#vertical,
zoom#reset100,zoom#popupslider]"
                        right="
[history#undo,history#redo,fullscreen#fullscreenexit]"/>
                    <replacementToolbars
jcr:primaryType="nt:unstructured">
                        <crop
                            jcr:primaryType="nt:unstructured"
                            left="[crop#identifier]"
                            right="
[crop#unlaunch,crop#confirm]"/>
                        <map
                            jcr:primaryType="nt:unstructured"
                            left="
[map#rectangle,map#circle,map#polygon]"
                            right="[map#unlaunch,map#confirm]"/>
                    </replacementToolbars>
```

```
                </fullscreen>
            </ui>
        </inplaceEditingConfig>
    </cq:inplaceEditing>
</jcr:root>
```

3. The unique aspect of the file is the `<parameters>` node that sets the resourceType to `wknd-spa-react/components/banner`.

```
<parameters
    jcr:primaryType="nt:unstructured"
    sling:resourceType="wknd-spa-react/components/banner"
    imageCrop=""
    imageMap=""
    imageRotate=""/>
```

Most component's do not require a `_cq_editConfig`. Image components and descendants are the exception.

Extend the Dialog

Our `Banner` component requires an extra text field in the dialog to capture the `bannerText`. Since we are using Sling inheritance, we can use features of the Sling Resource Merger to override or extend portions of the dialog. In this sample a new tab has been added to the dialog to capture additional data from an author to populate the Card Component.

1. In the `ui.apps` module, beneath the `banner` folder, create a folder named `_cq_dialog`.
2. Beneath `_cq_dialog` create a Dialog definition file `.content.xml`. Populate it with the following:

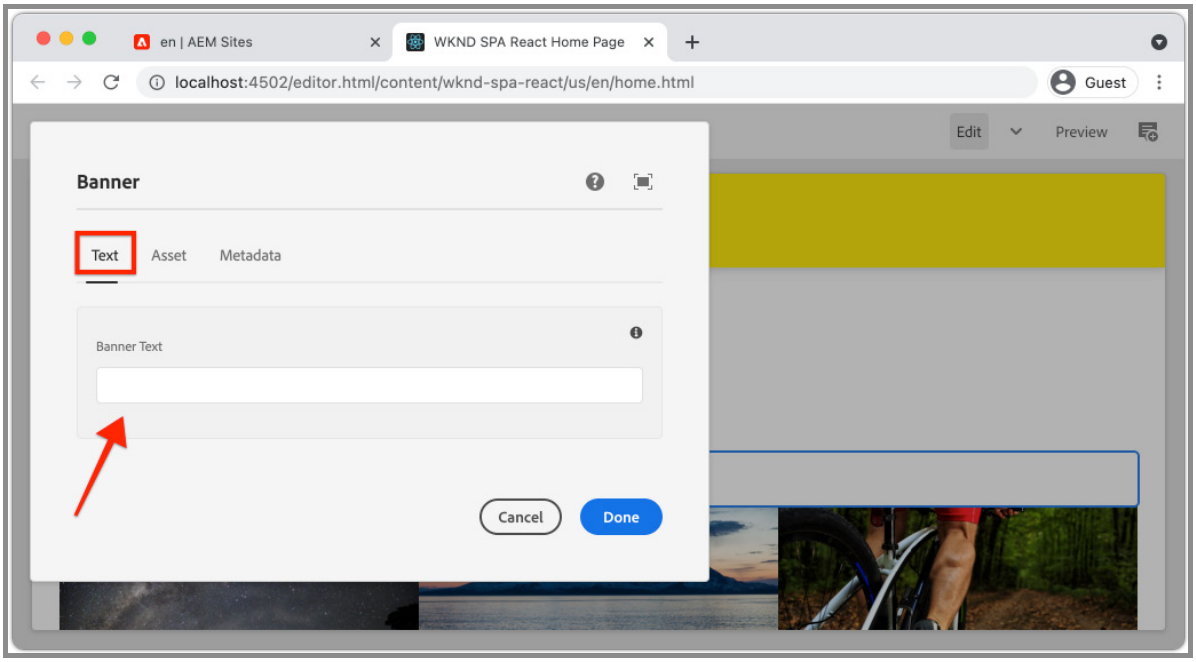
```
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:sling="http://sling.apache.org/jcr/sling/1.0"
xmlns:granite="http://www.adobe.com/jcr/granite/1.0"
xmlns:cq="http://www.day.com/jcr/cq/1.0"
xmlns:jcr="http://www.jcp.org/jcr/1.0"
xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
    jcr:primaryType="nt:unstructured"
    jcr:title="Banner"
    sling:resourceType="cq/gui/components/authoring/dialog">
    <content jcr:primaryType="nt:unstructured">
        <items jcr:primaryType="nt:unstructured">
            <tabs jcr:primaryType="nt:unstructured">
                <items jcr:primaryType="nt:unstructured">
                    <text
                        jcr:primaryType="nt:unstructured"
                        jcr:title="Text"
                        sling:orderBefore="asset"

 sling:resourceType="granite/ui/components/coral/foundation/conta
iner"
                        margin="{Boolean}true">
                        <items jcr:primaryType="nt:unstructured">
                            <columns
                                jcr:primaryType="nt:unstructured"

 sling:resourceType="granite/ui/components/coral/foundation/fixed
columns"
                                margin="{Boolean}true">
```

```
                                <items
jcr:primaryType="nt:unstructured">
                                    <column

 jcr:primaryType="nt:unstructured"

 sling:resourceType="granite/ui/components/coral/foundation/conta
iner">
                                        <items
jcr:primaryType="nt:unstructured">
                                            <textGroup

 granite:hide="${cqDesign.titleHidden}"

 jcr:primaryType="nt:unstructured"

 sling:resourceType="granite/ui/components/coral/foundation/well"
>
                                                <items
jcr:primaryType="nt:unstructured">
                                                    <bannerText

 jcr:primaryType="nt:unstructured"

 sling:resourceType="granite/ui/components/coral/foundation/form/
textfield"

 fieldDescription="Text to display on top of the banner."

 fieldLabel="Banner Text"

 name="./bannerText"/>
                                                </items>
                                            </textGroup>
                                        </items>
                                    </column>
                                </items>
                            </columns>
                        </items>
```

```
                    </text>
                </items>
            </tabs>
        </items>
    </content>
</jcr:root>
```

The above XML definition will create a new tab named **Text** and order it *before* the existing **Asset** tab. It will contain a single field **Banner Text**.

3. The dialog will look like the following:



Observe that we did not have to define the tabs for **Asset** or **Metadata**. These are inherited via the `sling:resourceSuperType` property.

Before we can preview the dialog, we need to implement the SPA Component and the `MapTo` function.

Implement SPA Component

In order to use the Banner component with the SPA Editor, a new SPA component must be created that will map to `wknd-spa-react/components/banner`. This will be done in the `ui.frontend` module.

1. In the `ui.frontend` module create a new folder for `Banner` at `ui.frontend/src/components/Banner`.

2. Create a new file named `Banner.js` beneath the `Banner` folder. Populate it with the following:

```
import React, {Component} from 'react';
import {MapTo} from '@adobe/aem-react-editable-components';

export const BannerEditConfig = {

    emptyLabel: 'Banner',

    isEmpty: function(props) {
        return !props || !props.src || props.src.trim().length <
1;
    }
};

export default class Banner extends Component {

    get content() {
        return <img
                classNam="Image-src"
                src={this.props.src}
                alt={this.props.alt}
                title={this.props.title ? this.props.title :
this.props.alt} />;
    }

    // display our custom bannerText property!
    get bannerText() {
        if(this.props.bannerText) {
            return <h4>{this.props.bannerText}</h4>;
```

```
        }

        return null;
    }

    render() {
        if(BannerEditConfig.isEmpty(this.props)) {
            return null;
        }

        return (
            <div className="Banner">
                {this.bannerText}
                <div className="BannerImage">{this.content}</div>
            </div>
        );
    }
}

MapTo('wknd-spa-react/components/banner')(Banner,
BannerEditConfig);
```

This SPA component maps to the AEM component `wknd-spa-react/components/banner` created earlier.

3. Update `import-components.js` at `ui.frontend/src/components/import-components.js` to include the new `Banner` SPA component:

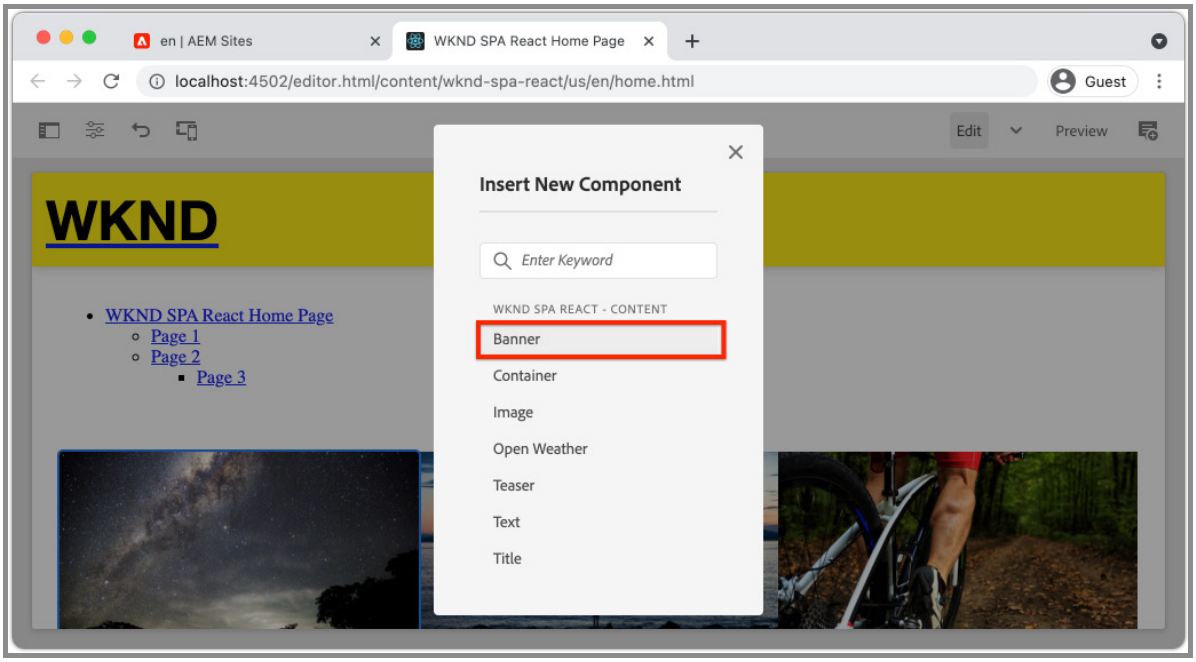
```
  import './ExperienceFragment/ExperienceFragment';
  import './OpenWeather/OpenWeather';
+ import './Banner/Banner';
```

4. At this point the project can be deployed to AEM and the dialog can be tested. Deploy the project using your Maven skills:

```
$ cd aem-guides-wknd-spa.react
$ mvn clean install -PautoInstallSinglePackage
```

5. Update the SPA Template's policy to add the `Banner` component as an **allowed component**.

6. Navigate to a SPA page and add the `Banner` component to one of the SPA pages:



NOTE:

The dialog will allow you to save a value for **Banner Text** but this value is not reflected in the SPA component. To enable, we need to extend the Sling Model for the component.

Add Java Interface

To ultimately expose the values from the component dialog to the React component we need to update the Sling Model that populates the JSON for the `Banner` component. This will be done in the `core` module that contains all of the Java code for our SPA project.

First we will create a new Java interface for `Banner` that extends the `Image` Java interface.

1. In the `core` module create a new file named `BannerModel.java` at `core/src/main/java/com/adobe/aem/guides/wkndspa/react/core/models`.

2. Populate `BannerModel.java` with the following:

```
package com.adobe.aem.guides.wkndspa.react.core.models;

import com.adobe.cq.wcm.core.components.models.Image;
import org.osgi.annotation.versioning.ProviderType;

@ProviderType
public interface BannerModel extends Image {

    public String getBannerText();

}
```

   This will inherit all of the methods from the Core Component `Image` interface and add one new method `getBannerText()`.

Implement Sling Model

Next, implement the Sling Model for the `BannerModel` interface.

1. In the `core` module create a new file named `BannerModelImpl.java` at `core/src/main/java/com/adobe/aem/guides/wkndspa/react/core/models/impl`.

2. Populate `BannerModelImpl.java` with the following:

```
package com.adobe.aem.guides.wkndspa.react.core.models.impl;

import
com.adobe.aem.guides.wkndspa.react.core.models.BannerModel;
import com.adobe.cq.export.json.ComponentExporter;
import com.adobe.cq.export.json.ExporterConstants;
import com.adobe.cq.wcm.core.components.models.Image;
import org.apache.sling.models.annotations.*;
import org.apache.sling.api.SlingHttpServletRequest;
import org.apache.sling.models.annotations.Model;
import org.apache.sling.models.annotations.injectorspecific.Self;
import
org.apache.sling.models.annotations.injectorspecific.ValueMapValu
e;
import org.apache.sling.models.annotations.via.ResourceSuperType;

@Model(
    adaptables = SlingHttpServletRequest.class,
    adapters = { BannerModel.class,ComponentExporter.class},
    resourceType = BannerModelImpl.RESOURCE_TYPE,
    defaultInjectionStrategy = DefaultInjectionStrategy.OPTIONAL
    )
@Exporter(name = ExporterConstants.SLING_MODEL_EXPORTER_NAME,
extensions = ExporterConstants.SLING_MODEL_EXTENSION)
public class BannerModelImpl implements BannerModel {

    // points to the the component resource path in ui.apps
    static final String RESOURCE_TYPE = "wknd-spa-
react/components/banner";

    @Self
```

```
    private SlingHttpServletRequest request;

    // With sling inheritance (sling:resourceSuperType) we can
adapt the current resource to the Image class
    // this allows us to re-use all of the functionality of the
Image class, without having to implement it ourself
    // see https://github.com/adobe/aem-core-wcm-
components/wiki/Delegation-Pattern-for-Sling-Models
    @Self
    @Via(type = ResourceSuperType.class)
    private Image image;

    // map the property saved by the dialog to a variable named
`bannerText`
    @ValueMapValue
    private String bannerText;

    // public getter to expose the value of `bannerText` via the
Sling Model and JSON output
    @Override
    public String getBannerText() {
        return bannerText;
    }

    // Re-use the Image class for all other methods:

    @Override
    public String getSrc() {
        return null != image ? image.getSrc() : null;
    }

    @Override
    public String getAlt() {
        return null != image ? image.getAlt() : null;
    }

    @Override
    public String getTitle() {
        return null != image ? image.getTitle() : null;
```

```
    }
```

```
    // method required by `ComponentExporter` interface
    // exposes a JSON property named `:type` with a value of `wknd-spa-react/components/banner`
    // required to map the JSON export to the SPA component props via the `MapTo`
    @Override
    public String getExportedType() {
        return BannerModelImpl.RESOURCE_TYPE;
    }

}
```

Notice the use of the `@Model` and `@Exporter` annotations to ensure the Sling Model is able to be serialized as JSON via the Sling Model Exporter.

`BannerModelImpl.java` uses the [Delegation pattern for Sling Models](https://github.com/adobe/aem-core-wcm-components/wiki/Delegation-Pattern-for-Sling-Models) to avoid rewriting all of the logic from the Image core component.

1. Observe the following lines:

```
@Self
@Via(type = ResourceSuperType.class)
private Image image;
```

The above annotation will instantiate an Image object named `image` based on the `sling:resourceSuperType` inheritance of the `Banner` component.

```
@Override
public String getSrc() {
    return null != image ? image.getSrc() : null;
}
```

It is then possible to simply use the `image` object to implement methods defined by the `Image` interface, without having to write the logic ourselves. This technique is used for `getSrc()`, `getAlt()` and `getTitle()`.

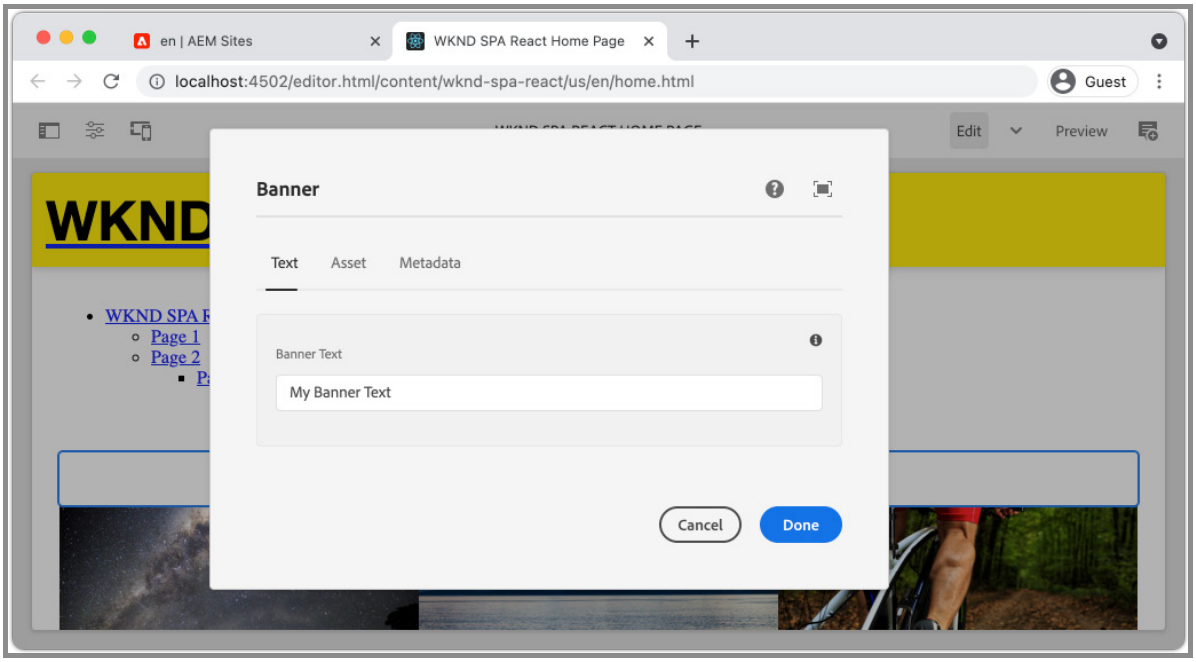
2. Open a terminal window and deploy just the updates to the `core` module using the Maven `autoInstallBundle` profile from the `core` directory.

```
$ cd core/
$ mvn clean install -PautoInstallBundle
```

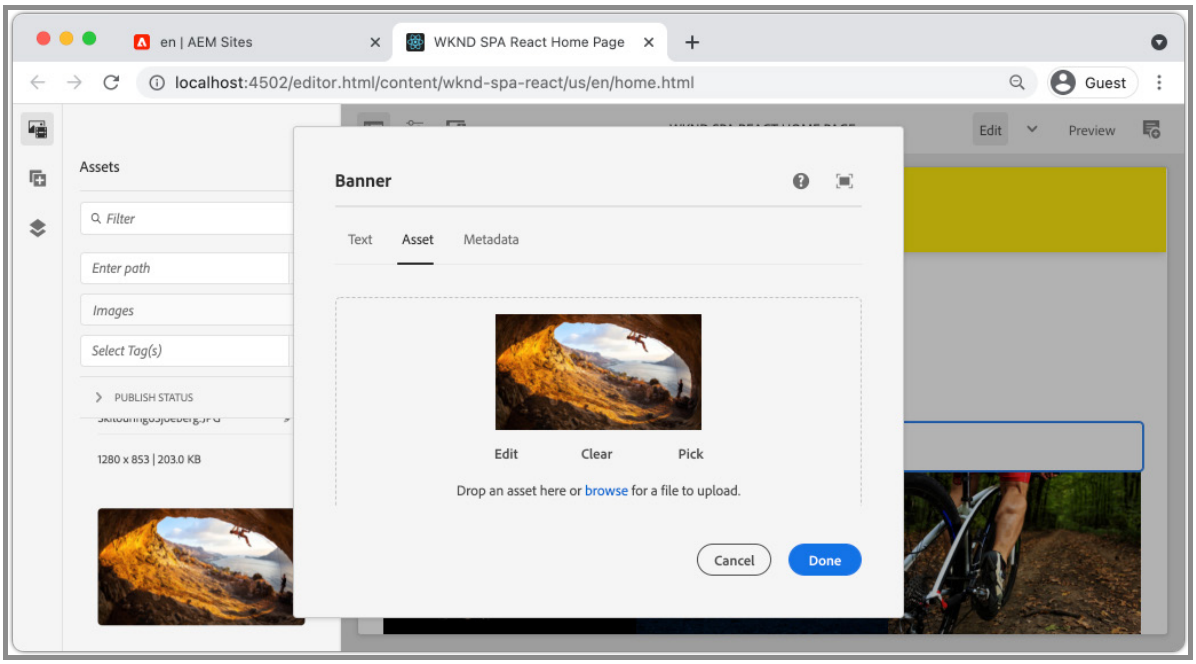
Putting it All Together

1. Return to AEM and open the SPA page that has the `Banner` component.

2. Update the `Banner` component to include **Banner Text**:

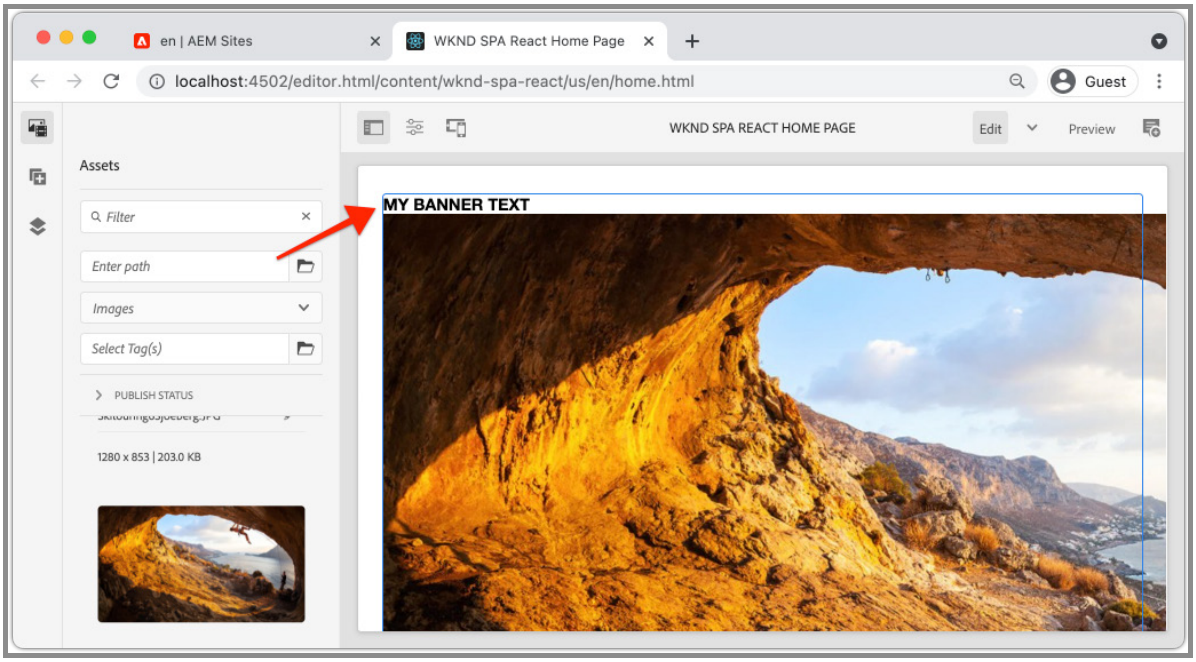


3. Populate the component with an image:



Save the dialog updates.

4. You should now see the rendered value of **Banner Text**:



5. View the JSON model response at: http://localhost:4502/content/wknd-spa-react/us/en.model.json and search for the `wknd-spa-react/components/card`:

```
"banner": {
    "bannerText": "My Banner Text",
    "src": "/content/wknd-spa-
react/us/en/home/_jcr_content/root/responsivegrid/banner.coreimg.
jpeg/1622167884688/sport-climbing.jpeg",
    "alt": "alt banner rock climber",
    ":type": "wknd-spa-react/components/banner"
 },
```

   Notice the JSON model is updated with additional key/value pairs after implementing the Sling Model in `BannerModelImpl.java`.

Congratulations, you learned how to extend an AEM component using the and how Sling Models and dialogs work with the JSON model.