LAB: Episerver as Headless

In this lab, you'll learn to make API calls to the Episerver Content API by building a small user interface for displaying content about a fictional music festival. The Episerver site and associated content have already been setup, and the Episerver Content API has been appropriately installed and configured to serve the relevant content.

You'll be writing Javascript to make API calls, but you'll be provided with code snippets to complete all functionality, so only a basic understanding of Javascript is required. You won't need to write any markup or HTML as that is provided for you.

Requirements

In order to complete this Lab, you'll need the following:

1. A code editor (Visual Studio Code works great - but whatever works best for you)
2. Google Chrome (or a similar browser with capable developer tools for inspecting requests)
3. Access to an Episerver instance with the Content API installed, and the Music Festival content added.

Activity 0: Setup

Goal

The goal of this activity is to setup your development environment, and gain a basic understanding of Vue.js before diving into Lab activities.

Instructions

1. Open the index.html file in a web browser

You should see simple page with the text "Welcome to the Music Festival lab". If this isn't showing up, ask for help or check the console for errors. During the lab, as you make updates to your code, you'll refresh this page to see the results.

2. Open Chrome Developer Tools by right clicking the page and selecting "Inspect"

You won't see much for now, but we'll use the Network and Console tabs to view requests we make to the Content API and debug.

3. Using your favorite code editor, open the index.js file

This lab uses Vue.js, a simple javascript framework often used for building single page applications. Within this file, you'll see that Vue is being initialized with a few things.

- A data object with a set of empty properties - we'll refer to this as our view model going forward. As we update properties on our view model from code, our view within index.html will automatically update.
- A created function - you can think of this as a constructor. When our Vue instance is created, this

function will be called. You'll see it is calling getSiteInfo and getArtists functions - we'll wire up and explain those later.

- A methods object - this contains functions that we will wire up to build our page. The getSiteInfo, getStartPage, getInfoPageContent, and getArtists functions will load information from the Content API in various ways by the end of the lab.

4. Update the title property of the view model to change it from "Welcome to the Music Festival lab" to be your name.

5. Reload index.html within the browser, and you should see your name displayed in the hero area of the page.

6. You're ready to dive in.

Activity 1: Establish an entry point with the Content API

Goal

The goal of this activity is to learn to use the Content API to locate the Start Page for a site, load the

Instructions

1. In index.js, add an API call to the Site Definition API within the getSiteInfo function. In order to make the request, we'll use a simple framework called axios which is already referenced in the project:

```
var vm = this;

axios.get(vm.apiUrl + '/episerver/site/', {
    headers: {
        'Accept': 'application/json'
    }
})
    .then(function (response) {
        console.log(response.data);

    })
    .catch(function (error) {
        console.log(error);
    });
```

2. Reload your browser, and open Developer Tools. On the Network tab under XHR, you should see a request to the Site Definition API. Click on it, and select the Preview tab in order to inspect the response. You will see that only one site is returned, with associated ContentRoot and Language information.

3. In index.js, update the then function of the code you added in Step 1 to locate and store the Start Page reference from the API response. To do this, we'll grab the first (and only) site from the array, locate the StartPage property within the ContentRoots object, and store it's Id property within our view model.

Finally, we'll call the getStartPage function, which we'll wire up later.

```
    //Grab the first site in the list
    var site = response.data[0];

    //Store the Start Page ID, and call getStartPage
    vm.startPageId = site.ContentRoots.StartPage.Id;
    console.log('Stored startPageId: ' + vm.startPageId);
    vm.getStartPage();
```

4. Reload your browser, and open the Console. You should see Stored startPageId: 5 printed.
5. Next, we'll load the actual data for the Start Page within the getStartPage function using our stored startPageId, and grab information from a ContentReference property. In index.js, add an API call to the Content API to load the Start Page in English (language code en), and store the MusicFestivalInfoPage reference ID within our view model.

```
    var vm = this;

    axios.get(vm.apiUrl + '/episerver/content/' + vm.startPageId, {
        headers: {
            'Accept': 'application/json',
            'Accept-Language': 'en'
        }
    })
        .then(function (response) {
            console.log(response.data);

            //Store the MusicFestivalInfoPage property as infoPageId, and call
getInfoPageContent
            var startPage = response.data;
            vm.infoPageId = startPage.MusicFestivalInfoPage.Value.Id;
            console.log('Stored infoPageId: ' + vm.infoPageId);
            vm.getInfoPageContent();
        })
        .catch(function (error) {
            console.log(error);
        });
```

6. Reload your browser, and open the Developer Console. You should see Stored infoPageId: 107 printed.

Activity 2: Load page content

Goal

The goal of this activity is to use the entry point we established in Activity 1 to load a single page from the Content API, and write code to display our content on the page.

Instructions

1. In index.js, add an API call to the Content API within the getInfoPageContent function using our stored infoPageId

```
    var vm = this;

    axios.get(vm.apiUrl + '/episerver/content/' + vm.infoPageId, {
        params: {

        },
        headers: {
            'Accept': 'application/json',
            'Accept-Language': 'en'
        }
    })
        .then(function (response) {
            console.log(response.data);

        })
        .catch(function (error) {
            console.log(error);
        });
```

2. Reload your browser, and open Developer Tools. On the Network tab under XHR, you should see the request to load the Info Page. Click on it, and select the Preview tab in order to inspect the response. You will see various standard Episerver properties, as well as properties specific to the InfoPage content type.

3. In index.js, update your call within getInfoPageContent to grab the Title, Preamble and MainBody properties from the response in the then function and update their values on the view model

```
    var page = response.data;

    //Set title, preamble, and main body properties from the loaded data
    vm.title = page.Title.Value;
    vm.preamble = page.Preamble.Value;
    vm.mainBody = page.MainBody.Value;
```

4. Reload your browser, you should see the Title, Preamble, and MainBody properties displayed on the page once the API call completes.

5. Next, we'll add a background image for our hero content. The image we're looking for is within the HeroContentArea property of our page, but if you look at the response from the API, you won't find the the Url of the image because, by default, the Content Api returns references for items within a Content Area. To gain access to the Url, let's add an expand property on the params object to ask that the

Content Api return the contents of the HeroContentArea in the same request:

```
    params: {
        expand: 'HeroContentArea'
    },
```

6. Reload your browser, and open Developer Tools to inspect the request again. If you look at the HeroContentArea property, you should see an ExpandedValue array which contains a full response object for each item referenced in the Content Area. This specific Content Area contains a single ImageData instance, with a background image.

7. Wire up the heroImage property of our view model using the Url of the image contained within the Content Area.

```
    //Grab the hero image from the first item in the HeroContentArea
    //expand parameter must indicate HeroContentArea to be able to access
ExpandedValue
    vm.heroImage = page.HeroContentArea.ExpandedValue[0].Url;
```

8. Reload your browser, and you should see a a background image in the Hero area of the page.

Activity 3: Add multi-language support

Goal

The goal of this activity is to use the Language information returned from the Content API to create a language switcher on our page. This will allow us to fetch content in multiple languages from the Content API.

Instructions

1. In index.js update the getSiteInfo function to utilize the Languages returned in the Site Definition API call you crated in Activity 1 to wire up a language switcher. To make the language switcher operate, we'll push all languages returned from the Site Definition API call into the languages property on our view model. Add the following code below the vm.getStartPage() function call within thegetSiteInfo function.

```
    //Update the languages data property on our view model
    site.Languages.forEach(function (language) {

        var item = {
            text: language.DisplayName,
            value: language.Name
        }

        vm.languages.push(item);
    });
```

2. Reload your browser, and a language switcher will appear at the top right of the page. You'll notice that English and Swedish options are available, based on the configured languages that are returned in the Site Definition API. However, the language switcher won't work just yet.

3. The language switcher works by binding and updating the currentLanguage property whenever the value of the dropdown is changed, and then re-calling the API calls we made in Activities 1 and 2. In order to wire everything up, we need to update our API calls to ensure we pass in the value of currentLanguage each time we make an API call. To do this, update the Accept-Language header used in the existing API calls within the getInfoPageContent andgetStartPage functions like this:

Old Value:

```
    'Accept-Language': 'en'
```

New Value:

```
    'Accept-Language': vm.currentLanguage
```

4. Reload your browser, and change the language to Swedish, and the content on the page should update accordingly.

5. Open Developer Tools, and change the language back and forth between Swedish and English. On the Network tab, inspect the API requests that are made, noting their Accept-Language header is different each time you change the language.

Activity 4: Search for artists

Goal

The goal of this activity is to gain an understanding of the Search API by building a grid of Music Festival artists on the page. You'll learn how to filter and sort content returned form the the Search API

Instructions

1. In index.js, add the following code to the getArtists function to make an API call to the Search API in order to fetch data about artists for the music festival from Episerver Find. In this call, we'll utilize the filter parameter to only grab content of type ArtistPage. In addition, we'll instruct the Search API to return the first 50 results using the top parameter. Once the API call returns, we'll grab the Results property of the data response, and update the artists property on our view model.

```
    vm.loadingArtists = true;

    axios.get(vm.apiUrl + '/episerver/search/content/', {
        params: {
            'top': 50,
            'filter': "ContentType/any(t:t eq 'ArtistPage')"
        },
        headers: {
            'Accept': 'application/json',
            'Accept-Language': vm.currentLanguage
        }
    })
        .then(function (response) {
            console.log(response.data);

            //Grab the artists from the Results property
            vm.artists = response.data.Results;

            vm.loadingArtists = false;
        })
        .catch(function (error) {
            vm.loadingArtists = false;
            console.log(error);
        });
```

2. Reload your browser, and you should see a grid of artists appear below the info page content. Open Developer Tools, and inspect the request made to the Search API, noting the structure of the request and response. In addition to the Results property containing data which matches our filter, there is also a TotalMatching property indicating the total items in the index which match our filter. In our case, the total number of artists is less then the top parameter, so we received all artists within Episerver.

3. Next, let's add some sorting capabilities to our Artist Grid. To do this, we'll add a orderby parameter to our API call, and bind it to the currentSort property of our view model like this:

```
        params: {
            'top': 50,
            'filter': "ContentType/any(t:t eq 'ArtistPage')",
            'orderby': vm.currentSort
        },
```

4. To finish wiring up our sort options, let's add a list of potential sort options to our view model. In the view model, update the sortOptions property to provide a variety of sorting options to the user. In addition, set the default sort to ArtistName/Value asc by updating the currentSort property. Update the values below:

Old Value:

```
    currentSort: '',
    sortOptions: [

    ]
```

New Value:

```
    currentSort: 'ArtistName/Value asc',
    sortOptions: [
        {
            text: "Artist Name",
            value: "ArtistName/Value asc"
        },
        {
            text: "Performance Time",
            value: "PerformanceStartTime/Value asc"
        },
        {
            text: "Headliner",
            value: "ArtistIsHeadliner/Value desc, ArtistName/Value asc"
        },
        {
            text: "Genre",
            value: "ArtistGenre/Value, ArtistIsHeadliner/Value desc"
        },
    ]
```

5. Re-load the page, and note the new sort dropdown with the options bound to the values above. Update the value of the sort dropdown, and note the change in order of artists within the grid. Open Developer Tools, and as you update the sort dropdown, note the requests made back to the Search API, and the format of the orderby parameter.

6. Let's add filtering. It will work similarly to our dynamic sorting, with another dropdown for the user to adjust what shows up in the Artist grid. To accomplish this, we'll first make the filter parameter in our Search API call dynamic. Update the params object to pass vm.currentFilter to the filter property instead of the current value.

Old Value:

```
        params: {
            'top': 50,
            'filter': "ContentType/any(t:t eq 'ArtistPage')",
            'orderby': vm.currentSort
        },
```

New Value:

```
         params: {
             'top': 50,
             'filter': vm.currentFilter,
             'orderby': vm.currentSort
         },
```

7. Next, let's add a list of potential filter options to our view model. In the view model, update the filterOptions property to provide a variety of filtering options to the user. In addition, set the default filter to ContentType/any(t:t eq 'ArtistPage') by updating the currentFilter property. Update the values below:

Old Value:

```
    currentFilter: "",
    filterOptions: [

    ],
```

New Value:

```
    currentFilter: "ContentType/any(t:t eq 'ArtistPage')",
    filterOptions: [
        {
            text: "All",
            value: "ContentType/any(t:t eq 'ArtistPage')"
        },
        {
            text: "Headliners",
            value: "ContentType/any(t:t eq 'ArtistPage') and
ArtistIsHeadliner/Value eq true"
        },
        {
            text: "Saturday",
            value: "ContentType/any(t:t eq 'ArtistPage') and
PerformanceStartTime/Value lt 2018-04-01T03:00:00Z"
        },
        {
            text: "Sunday",
            value: "ContentType/any(t:t eq 'ArtistPage') and
PerformanceStartTime/Value gt 2018-04-01T03:00:00Z"
        },
    ],
```

8. Re-load the page, and note the new filter dropdown. As you change the value of the dropdown, the contents of the Artist Grid will change. Open Developer Tools, and inspect the requests made back to the Search API, noting the various filter options employed via the filter parameter.

9. Brainstorm additional sorting and filtering options and try to add them to the filterOptions and sortOptions arrays to provide additional functionality to the user. Perhaps, a filter for each Stage?