Lesson 28: Rendering Custom HTML

Created by Jacques Marais, last modified on Nov 23, 2018

> *As a shop owner I want to have a visual summary of the sales for my shops.*

Lesson Outcomes

By the end of this lesson you should:

- Know how to package custom static content sources files with a DSL app
- Know how to use the `<static>` view components to render static content on a DSL view
- Know how custom static content is integrated with an app using the inbound API
- Know how to use additional resources, including external libraries, in a custom static content view component

App Use Case Scenario

A shop owner should have access to a dashboard or a set of views that provides a visual representation of the sales for his shops. This should include a visual summary of the total sales value per product, the total sales quantity per product and the value of sales per farmer. The first two visualizations should make use of pie charts and the last visualization should make use of a column chart.

To achieve this we will need to add the relevant static content source files, the relevant Helium view that will display the static content and any inbound APIs that will provide data for the visual representations.

New & Modified App Files

`./services/ApiPurchaseResource.mez`

`./web-app/lang/en.lang`

`./web-app/presenters/purchases/PurchasesReports.mez`

`./web-app/static/shopowner_dashboard/shopowner_dashboard.css`

`./web-app/static/shopowner_dashboard/shopowner_dashboard.html`

`./web-app/static/shopowner_dashboard/shopowner_dashboard.js`

`./web-app/views/purchase_dashboard/PurchaseDashboard.vxml`

`./web-app/views/purchase_dashboard/PurchaseFrequency.vxml`

Custom HTML Hello World

Before the tutorial app use case, as described above, is addressed this lesson first covers a basic "Hello World" example of rendering custom static content on a DSL app view. Following this, some additional topics are covered regarding the inclusion of custom static content in DSL apps. Once these basic topics have been covered the tutorial app use case is addressed.

Packaging Static Content Source Files

Static content files should be included in web-app/static, be of type css, js, png, jpeg, svg, htm or html and file names must be unique across the whole app.

In order to package static content with a DSL app, it should be included in the **web-app/static** folder. Any recursive folder structure can be used within this folder but the file names should be unique as only the file name will be used by Helium to reference the static content resources and not the file path. Static content includes files with file extension **css**, **js**, **png**, **jpeg**, **svg**, **htm** or **html**. For this example consider the following html and JavaScript content in a file named **hello_world.html**.

```
<div>
    <h2>Hello World!!</h2>
    <p id='dayOfWeek'></p>
    <script>
        var d = new Date();
        var weekdays = new Array(7);
        weekdays[0] =  "Sunday";
        weekdays[1] = "Monday";
        weekdays[2] = "Tuesday";
        weekdays[3] = "Wednesday";
        weekdays[4] = "Thursday";
        weekdays[5] = "Friday";
        weekdays[6] = "Saturday";

        document.getElementById('dayOfWeek').innerHTML = 'Today i
    </script>
</div>
```

Note how the above represents a HTML partial contained within a div tag. Also note the use of JavaScript as contained inside the `<script>` tag.

All content for the "Hello World" example described in this section is simply shown in this lesson as an example and is not included with the lesson's source code.

Helium View Component

Now that we have our initial static content resource represented by an HTML partial, we need to reference it from an app view. For this example consider the following view:

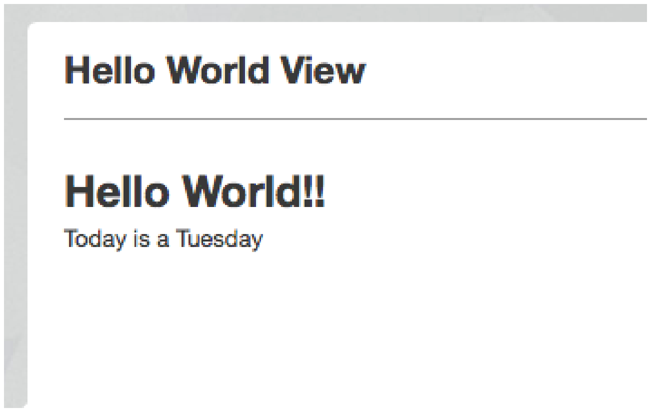
```
 <view label="view_heading.hello_world">

    <menuitem label="menu_item.hello_world">
        ...
    </menuitem>

    <static source="hello_world.html"/>
</view>
```

The source attribute in our `<static>` view component references the **hello_world.html** resource in the app source code. The following validations apply to the

`<static>` view component as shown above:

- A value has to specified for the source attribute
- The value specified for the source attribute has to be a valid static content source file packaged with the app source and it should be of type **`html`** or **`htm`**.
- If the `<static>` component is used on a view, no other widgets. Row actions are allowed.

From the example above, the following is rendered on the view in the Helium app:


Hello World View

Hello World!!

Today is a Tuesday

Using Additional Resources Inside an HTML Partial

Helium provides functionality for developers to reference additional static content resources from their rendered HTML partial. This is achieved by using a jQuery ajax call to a Helium REST resource specifically provided for the purpose of retrieving additional static content resources from within another static content resource. Below is an example of such an ajax call:

```
<script type="text/javascript">
    var baseUrl = '/web-api/services/app/exec/static/';
    var hsid = '?hsid=' + ${hsid};
    var jsUrl = baseUrl + 'my_js_resource.js' + hsid;
    $.ajax({
        url: jsUrl,
        dataType: 'json',
        cache: false,
        error: function (jqXHR, textStatus, errorThrown) {
            console.log("Failed to load JS resource: " + jsUrl +
            console.log(errorThrown);
        },
        success: function (data, textStatus, jqXHR) {
            // Decode result and execute JS
            eval(window.atob(data.data));
        }
    });
</script>
```

In the above example, we can see how the Helium provided rest resource, **`/web-api/services/app/exec/static/`**, is invoked by referencing the required static content resource by file name as a path parameter.

Helium uses the session id of the currently logged in user for authentication. This session is represented by a Helium session id or `hsid`. The value of this `hsid` needs to be specified as a value for the `hsid` query parameter when invoking the above API. To improve usability for developers, this value can be specified as `${hsid}` in which case Helium will do a string replacement with the current `hsid` value as appropriate. Alternatively developers can reference `appConfig.hsid` to get the value as this is globally accessible to developers. It is however recommended that the templated value, `${hsid}`, is used.

Using The Inbound API With Custom Static Content

The inbound API, as discussed in tutorial lessons 25 and 27, is available to reference from any app static content resources. As with the Helium provided REST resource for static content resources as mentioned above, the inbound API for an app can be invoked using a jQuery ajax call.

Also, as with the above mentioned REST resource, the inbound API, when invoked from within the static content for an app, needs to be passed the `hsid` for the current user session. This is done by passing the value as query parameter. When invoking the inbound API in such a manner, it is invoked from the context of a current app being executed. This provides some advantages such as:

- All unit variable values for the current execution context of the app is available providing seamless integration with normal DSL views and custom static content views (by means of the inbound API).
- Context related to the currently logged in user is available to the inbound API functions. For example a call to `WhateverTheLoggedInRole:user()` will return a valid value instead of **`null`**.

It's important to take note of the differences when invoking the inbound API from within the static content of an app versus invoking the inbound API from an external client:

	From app static content	From external API client
Authentication	Authenticates against currently logged in user browser session	Authenticates against provided API credentials
Access to app context	Access to full app execution context such as unit variables, currently logged in user etc.	App execution context is not applicable as only the API function is executed and not the entire app.
Required query parameters	`hsid` query parameter needs to be specified. Allows Helium to be aware of the current app execution context and logged in user. See "Authentication" section above.	No built in query parameter values are required.

Accessing Unit Variables

Although unit variables are not directly accessible from any custom static content, they can accessed using the inbound API as described above. For example, adding a dedicated API function that returns the current unit variable values are allowed. This is because the API is invoked from within the context of the current app execution.

Using External Libraries

Similarly to invoking internal app or Helium provided API resources, external libraries can be also be imported using a jQuery ajax call. The JavaScript code snippet below shows, for example, how the google charts library can be imported and initialized:

```
// Load the library and kick off initializing thereof
$.ajax({
    url: "https://www.gstatic.com/charts/loader.js",
    dataType: 'script',
    success: initLibs,
    async: true
});

function initLibs() {
    // Load the Visualization API and the corechart package.
    google.charts.load('current', {'packages':['corechart']});

    // Set a callback to run when the Google Visualization API is
    google.charts.setOnLoadCallback(loadPurchaseValueReport);
}

function loadPurchaseValueReport() {
    ...
}
```

Implementing the Purchases Dashboard Use Case

At this stage of the lesson, we have discussed of all the topics and features required to implement the app use case as described in the beginning of this lesson. We can now look at each component that needs to be implemented for the app use case.

Backing Inbound API

The purchases dashboard will contain three charts to graphically represent data related to purchases. These charts will represent the total value of purchases per stock item, the total quantity of purchases per stock item and the value of purchases per farmer. We will need inbound API functions to back each one of these. We add the following to a newly introduced unit, `ApiPurchaseResource`, located in **`/services/ApiPurchaseResource.mez`** :

```
 // Returns a json object representing the value of purchases per
// for the shops related to the specified shop owner
@GET("v1/purchase/report/product/value")
json getProductPurchaseValueReport() {

    // Get the relevant purchases
    FarmerPurchase[] purchases = getLoggedInShopOwnerPurchases();
    if(purchases == null) {
        return null;
    }
    // Construct the json result
    json result = "{}";
    foreach(FarmerPurchase purchase: purchases) {
        Stock stockItem = purchase.stock;
        if(stockItem != null) {
            if(result.jsonGet(stockItem.name) == null) {
                result.jsonPut(stockItem.name, purchase.finalCost
            }

            if(result.jsonGet(stockItem.name) != null) {
                int currentPurchaseValue = result.jsonGet(stockIt
                result.jsonPut(stockItem.name, currentPurchaseVal
            }
        }
    }
    return result;
}
```

```
    // Returns a json object representing the quantity of purchases p
    // for the shops related to the specified shop owner
    @GET("v1/purchase/report/product/quantity")
    json getProductPurchaseQuantityReport(uuid shopOwnerId) {
        // Get the relevant purchases
        FarmerPurchase[] purchases = getLoggedInShopOwnerPurchases();
        if(purchases == null) {
            return null;
        }
        // Construct the json result
        json result = "{}";
        foreach(FarmerPurchase purchase: purchases) {
            Stock stockItem = purchase.stock;
            .
            .
            .
        }
        return result;
    }
```

```
     // Return a json object representing the total value of purchase
    // for shops belonging to the shop owner
    @GET("v1/purchase/report/farmer/value")
    json getFarerPurchaseValueReport(int numOfFarmers) {
        // Get the relevant purchases
        FarmerPurchase[] purchases = getLoggedInShopOwnerPurchases();
        if(purchases == null) {
            return null;
        }
        // Construct the json result
        json result = "{}";
        foreach(FarmerPurchase purchase: purchases) {
            Farmer farmer = purchase.farmer;
            .
            .
            .
        }
        return result;
    }
```

All three of the above inbound API functions make use of the following helper method:

```
    // Helper method to get the farmer purchases for all shops relate
```

```
// logged in shop owner
FarmerPurchase[] getLoggedInShopOwnerPurchases() {

    // Get the shop owner
    ShopOwner shopOwner = ShopOwner:user();
    if(shopOwner == null) {
        return null;
    }
    // Get the shops linked to the shop owner
    .
    .
    .

    // Get the relevant purchases
    .
    .
    .
    return purchases;
}
```

Note from the above code snippet how we make use of `ShopOwner:user()`. As mentioned before we can only do this because of the fact that the API is invoked from within the context of the app execution for the currently logged in user and not from an external client.

Static Content Source Files

Now that we have the backing inbound API, we can add the static content source files, representing the purchases dashboard, to our app. We will add three separate files to represent the html partial that will be reference from the DSL app view, the JavaScript content that will be referenced from our html partial and the css content that will also be referenced from the html partial to render.

The following files will be added to `/web-app/static/purchases_dashboard`:

shopowner_dashboard.html

```
 <div>
    <p id="data_par"></p>
    <script type="text/javascript">

        var baseUrl = '/web-api/services/app/exec/static/';
        var hsid = '?hsid=' + ${hsid};
        var jsUrl = baseUrl + 'shopowner_dashboard.js' + hsid;
        var cssUrl = baseUrl + 'shopowner_dashboard.css' + hsid;
        $.ajax({
            url: jsUrl,
            dataType: 'json',
            cache: false,
            error: function (jqXHR, textStatus, errorThrown) {
                console.log("Failed to load JS resource: " + jsUr
                console.log(errorThrown);
            },
            success: function (data, textStatus, jqXHR) {
                // Decode result and execute JS
                eval(window.atob(data.data));
            }
        });

        // Load CSS
        $.ajax({
            url: cssUrl,
            dataType: 'json',
            cache: false,
            error: function (jqXHR, textStatus, errorThrown) {
                console.log("Failed to load CSS resource: " + cssU
                console.log(errorThrown);
            },
            success: function (data, textStatus, jqXHR) {
                // Decode result and include styles
                $('head').append('<style type="text/css">' + wind
            }
        });

    </script>

    <div class="grid-container">
        <div id="salesValuePerStockItem">1</div>
        <div id="salesQuatityPerStockItem">2</div>
        <div id="purchasesPerFarmer">3</div>
    </div>
</div>
```

shopowner_dashboard.js Expand source

shopowner_dashboard.css

```
.grid-container {
  display: grid;
  grid-template-columns: auto auto;
}
```

View Rendering Static Content

The final component that needs to be added is the view that makes use of the `<static>` view component and reference our html partial as a custom static resource to render. For this we will also be refactoring the `PurchaseFrequency` view slightly:

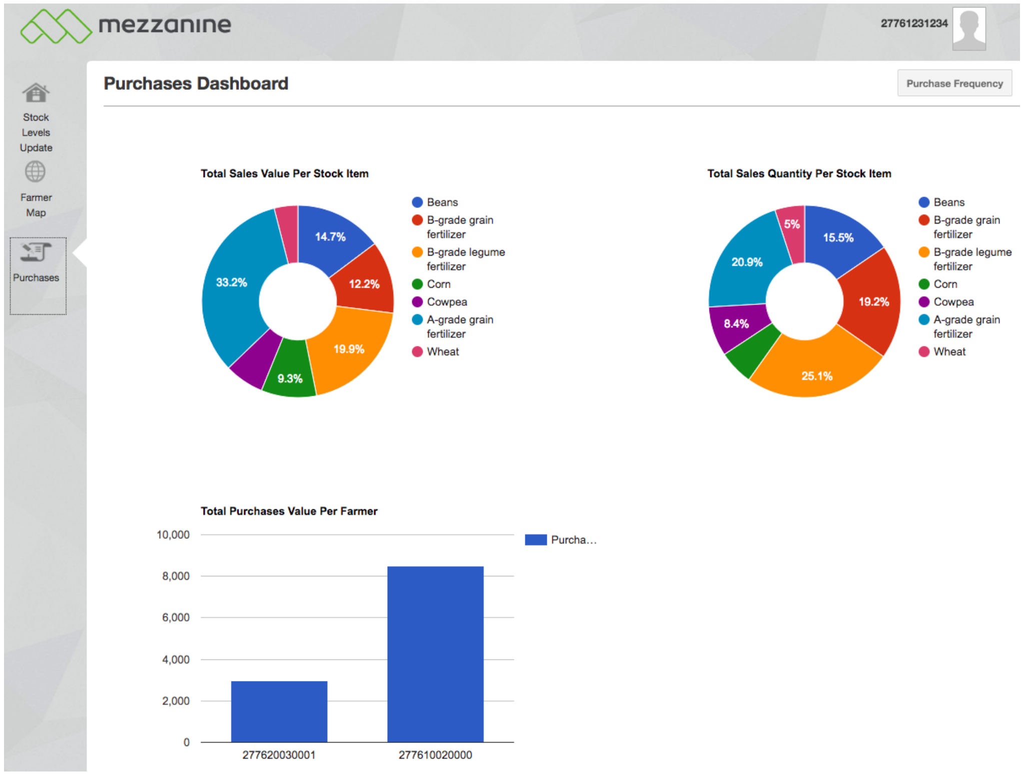
- We add a `PurchaseDashboard` view with a menu item labelled "Purchases". This view will be used to render our custom static content.
- We add a view action to the above mentioned view that navigates to the `PurchaseFrequency` view and a view action on the `PurchaseFrequency` view that navigates to the `PurchaseDashboard` view.
- We also remove the menu item from the `PurchaseFrequency` view.

The newly introduced `PurcahseDashboard` view has the following content:

```
<view label="view_heading.purchases_dashboard" unit="PurchasesRep
```

```
    <menuitem label="menu_item.purchases" icon="Admin">
        <userRole>Shop Owner</userRole>
    </menuitem>

    <action label="action.purchase_frequency" action="navigateToPu

    <static source="shopowner_dashboard.html"/>
</view>
```

The above results in the following being rendered:



Debugging Custom Static Content Issues

With the powerful ability to add custom static content to a Helium view, a whole range of additional issues can arise. These might be related to HTML, CSS, JavaScript or even the specific library that is being used. In order to resolve these issues as productively as possible it's advised that you consult any available online documentation regarding the relevant technologies. Sites like Stack Overflow is also very useful for resolving such issues as the likelihood that someone else, that is part of the community, has encountered a similar issue is high.

In addition, ensure that you are using all the features your browser provides to assist in web development. Resources such as the console and inspector of chrome devtools are invaluable.

If you are not able to resolve an issue using the above methods or you suspect that the issue is specifically related to Helium, feel free to follow the standard support process as described here.

Additional Resources

Additional resources on making use of the inbound app API:

- Lesson 25: Supporting Inbound REST API Post Functionality
- Lesson 27: Expanding Inbound REST API Functionality
- Inbound API Annotations Reference

- Native JSON Types Reference

Additional resources on rendering custom static content in Helium DSL apps:

- <static/> View Component Reference
- Helium Custom Web Content Reference

External resources:

- jQuery ajax documentation
- Getting started with google charts

Lesson Source Code

Lesson 28.zip

No labels