HCL Volt MX

Building a Volt MX App for Domino

Matthias Schneider – matthias.schneider@hcl.com

Table of Contents

Author

This document was created by the following Subject Matter Expert:

Matthias Schneider
Company: HCL

Bio

Matthias Schneider joined Lotus Professional Services in 2000. After developing for and managing Domino and WebSphere Portal environments for years, he focused on the rising IBM Connections solution. Today, he is working as an HCL Technical Advisor for HCL Connections, Volt MX and other HCL products.

Contact: matthias.schneider@hcl.com

Introduction

This manual is intended to be used by HCL employees, business partners and customers for familiarizing with the HCL Volt MX platform and implementing their first mobile app. The steps described here can be finished in 3-4 hours, so feel free to use it also for workshop and demo purposes.

Some of the aspects described here:

- You will build a Volt MX Foundry application that connects to an XML / JSON backend.
- You will implement services and operations retrieving data from several platforms, in our case HCL Domino and – optionally - Wikipedia.
- You will start from scratch with implementation of a mobile app in Volt MX Iris.
- You will learn how to make use of templates, build your own JavaScript libraries and re-use existing components from the HCL Volt MX Forge Marketplace.
- You will design and code page structures and actions within your mobile app.
- You will learn how to publish and test your mobile app.

Prerequisites

To complete this tutorial, you will need:

- Access to HCL Volt MX Foundry
- Access to HCL Domino and the PhotoBlog app
- Access to Volt MX Iris client
- A mobile device that has internet access
- HCL Volt MX app installed on your device plus HCL Cloud account (iOS only)

Note: This lab is an extension / next step for the official self-paced workshop provided via HCL Academy website. It could be used independently from that workshop with proper server setup, Domino database deployment and access to the required systems.

To start from scratch, run the following steps:

- Setup a Domino server that hosts HTTP services. The lab makes use of the Domino Access Services (DAS) JSON API, which could be enabled on even older Domino releases. If you plan for Volt MX Go, you can also start with Domino 12.0.2 and the new REST API.
- Enable DAS on your Domino server.
- Create a database from the given template. It should automatically have DAS enabled on database and design level.
- Get a Volt MX account plus Iris / Foundry. The easiest way to accomplishg that is by joining the free Volt MX Trial, but you can deploy Foundry locally too. Iris is an IDE that always needs to be installed locally on the developer's machine.
- See *References* for more details.

Use Case / Result

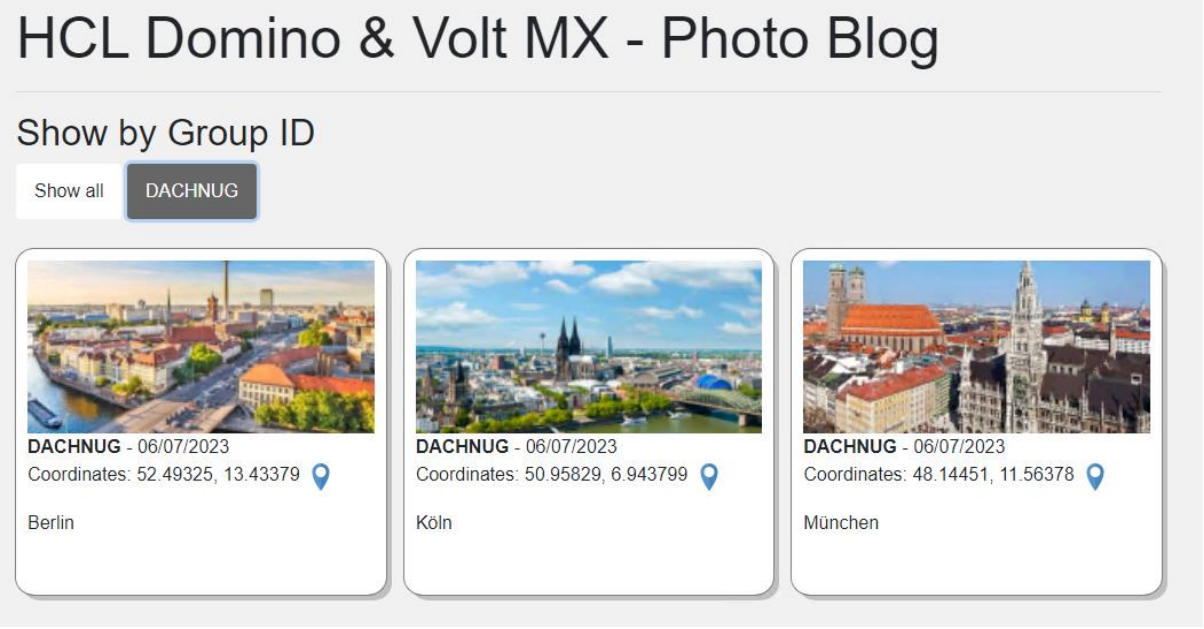
An organization has established a Domino application to plan and document travel to different locations.

In our scenario, this data source should be aggregated in a mobile application, to visualize recently visited locations in a map on a mobile device.

Reviewing backend integration

In our tutorial, we will create a mobile app that should connect to a backend system to retrieve and visualize data.

For the Domino part, we will connect to a database that hosts our travel information.



Note: If you have finished the self-paced workshop and want to continue directly from your stage, you can create a number of blog documents from Foundry that reflect different locations – it will make the location / navigation experience better. Write down the **blogGroupID** parameter you used (like "*DACHNUG"*), we will point to that later.

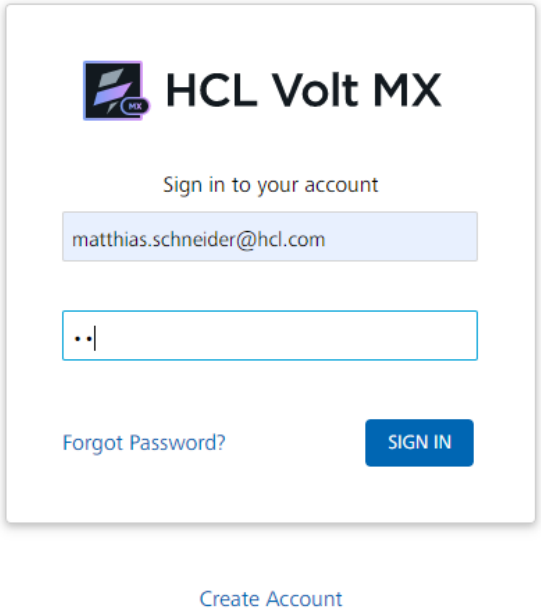
We will continue to use the Domino Access Services API to retrieve information from that database.


```
[
  {
      "@href":"\/photoblog.nsf\/api\/data\/collections\/name\/Posts\/unid\/387A7CFB5D45BAD5002589C7002BD8B0",
      "@link":
      {
          "rel":"document",
          "href":"\/photoblog.nsf\/api\/data\/documents\/unid\/387A7CFB5D45BAD5002589C7002BD8B0"
      },
      "@entryid":"1-387A7CFB5D45BAD5002589C7002BD8B0",
      "@unid":"387A7CFB5D45BAD5002589C7002BD8B0",
      "@noteid":"916",
      "@position":"1",
      "@siblings":3,
      "@form":"PhotoBlog",
      "blogGroupId":"DACHNUG",
      "blogDate":"2023-05-31",
      "UNID":"387A7CFB5D45BAD5002589C7002BD8B0",
      "blogLocation":"50.958296, 6.943799",
      "blogComment":"K\u00F6ln"
  },
  {
      "@href":"\/photoblog.nsf\/api\/data\/collections\/name\/Posts\/unid\/C43934C81F60CDBE002589C7002BBC9F",
```

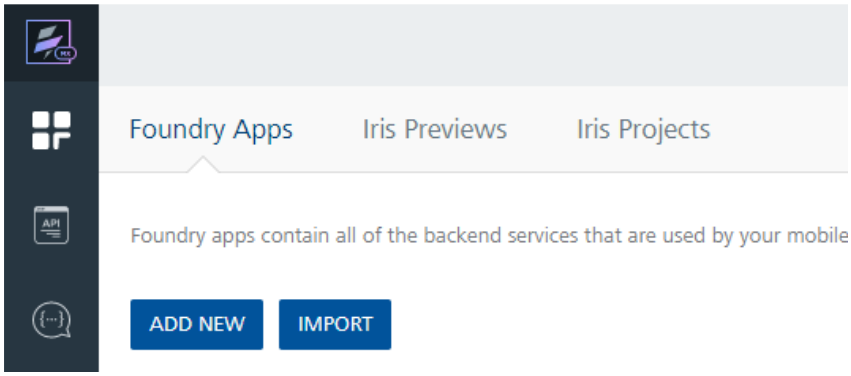
Building the Foundry project

To hook into the Domino API and retrieve data, we need to establish a channel that not only secures backend access but also allows us to filter the information before posting it to the app.

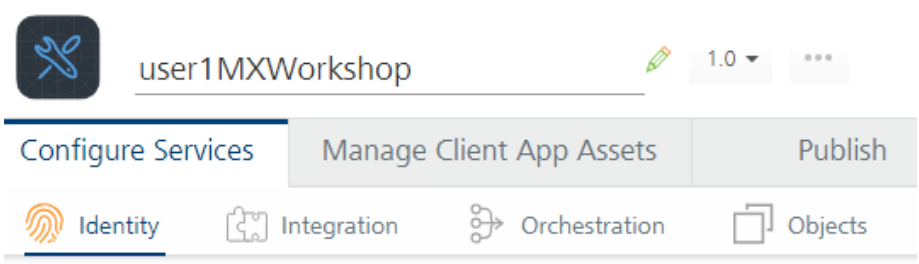
From your browser, connect to your Foundry backend and authenticate using your credentials:



Navigate to **Foundry Apps** and click on **ADD NEW**.



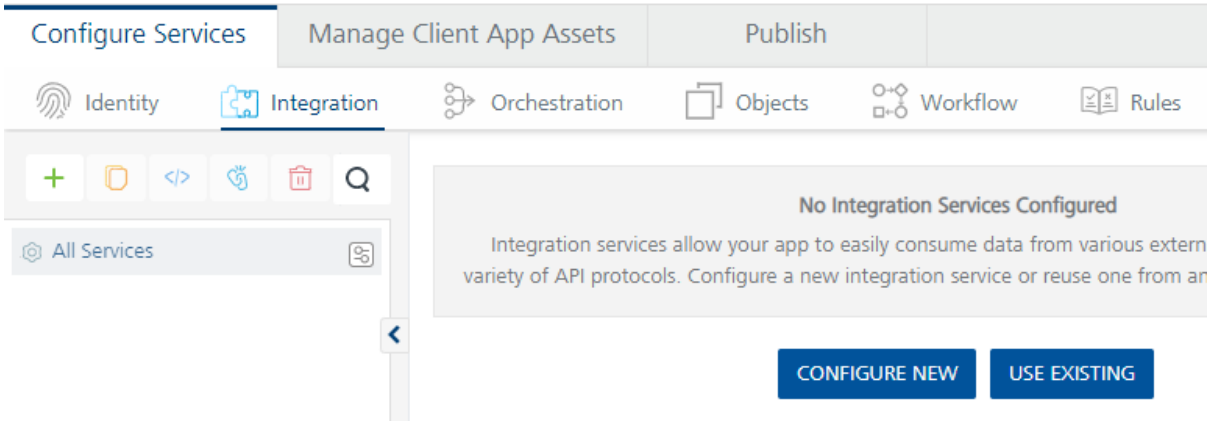
Click on the pencil icon next to the default name and change the project name to **<userX>MXWorkshop**.



Note: In our scenario, we will start building a set of **Integration** services that connects to HCL Domino, to retrieve data. Usually, you would first establish an **Identity** service for authenticating the user against an LDAP or other identity management system.

In our scenario, our set of **Integration** services will use Basic Authentication and therefore require proper authentication anyway as part of our services.

Navigate to the **Integration** tab and click on **CONFIGURE NEW**.

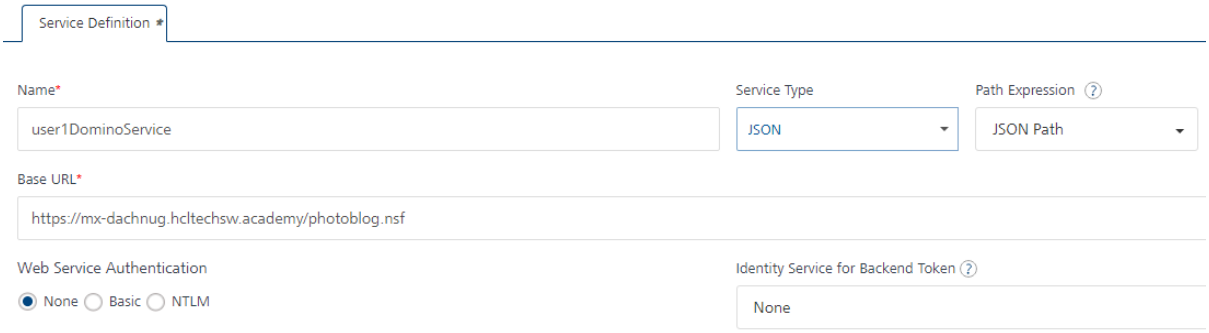


In our first step, we will create our Domino-based service and its operation. Name the Service **<userX>DominoService**. You can choose a different name, as long it is unique within the Foundry platform, contains 30 characters maximum and no special characters.

Note: You can access any Domino database, as described here, that matches the following requirements:

- **Data Services** (Domino Access Services) have been enabled on the Domino server
- Domino Access Services have been enabled for the Domino database (Views & Documents)
- Domino Access Services + Operations have been enabled for the Domino database views you'd like to access

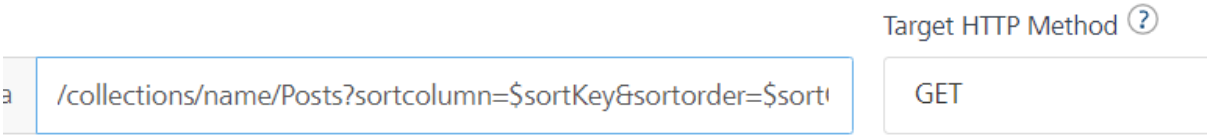
For our scenario, we use the **JSON** Service Type to access the following Base URL (the URL from screenshot below is just an example):
https://[yourDominoServer]/photoblog.nsf/api/data



Click **SAVE & ADD OPERATION**.

Name the new operation **GetBlogByGroupID** and make it available to **Anonymous App Users**.

URL suffix is
/collections/name/Posts?sortcolumn=$sortKey&sortorder=$sortOrder&search=FIELD%20blogGroupId=%22$blogGroupId%22&start=0&count=99.



Note: An **Operation Security Level** of **Authenticated App Users** would require authentication / authorization against an **Identity** service before. This service would then pass a token to the **Integration** service, confirming the identity of the app user.

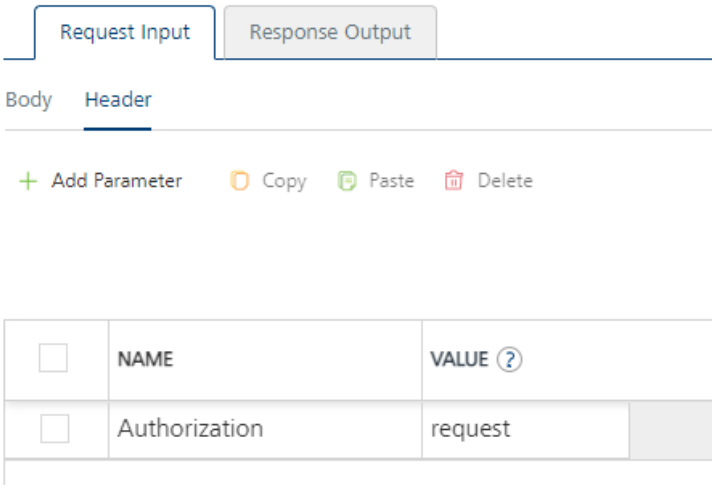
In our scenario, we will switch that setting to **Anonymous App Users**, meaning that authentication will be part of the **Integration** service' API call. We will authenticate our service requests via Basic Authentication.

We will now add the 3 placeholders **blogGroupId**, **sortOrder** (TEST / DEFAULT **descend**) and **sortKey** (TEST / DEFAULT **blogDate**) as **Request Input Body** parameters.

NAME	VALUE	TEST VALUE	DEFAULT VALUE	DATA TYPE
blogGroupId	request	DACHNUG	DACHNUG	string
sortOrder	request	descend	descend	string
sortKey	request	blogDate	blogDate	string

For the group id, use "*DACHNUG*". This is more less a category assigned to your documents, to separate your documents from another user's photo blogs.

Navigate to **Request Input > Header** and add a parameter **Authorization**.

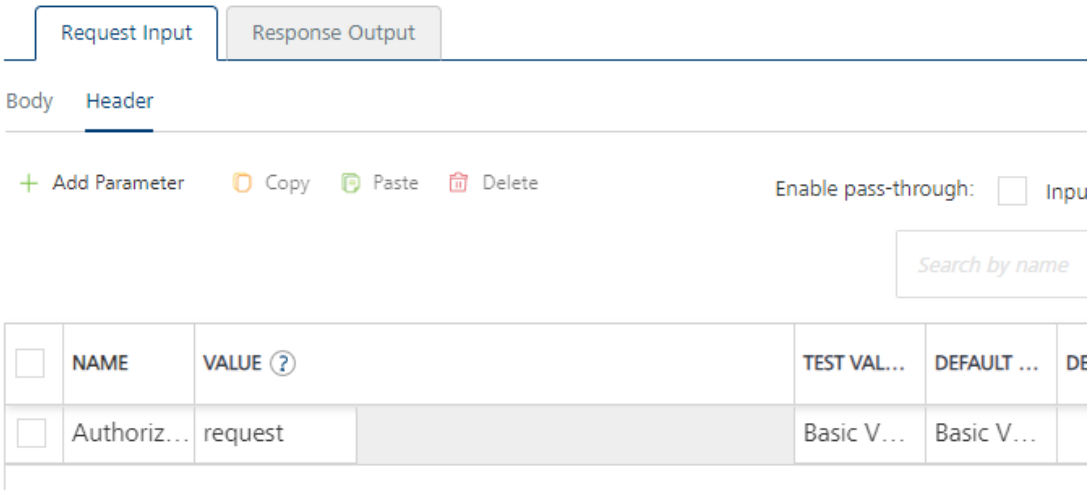


For Basic Authentication we need to pass the value **Basic**, followed by the base64-encoded combination of **<username>:<password>**.

Pages like **https://www.base64encode.org** can do that job for you.

Back in the Foundry Service Operation, enter **Basic** followed by a blank and the resulting Base64 code as a **TEST VALUE** here.

Example: ***Basic ZHVzZXI6Vm9sdEAxMjM0***



Note: If you enter a **TEST VALUE** here, this impacts the Foundry application only. It allows you to test the service call from within the Foundry app. Any call from there will use the credentials defined here.

If you enter a **DEFAULT VALUE** here, this impacts both the Foundry application AND the Iris app. The credentials defined here will be used if the mobile app connects to the service and does not send an authorization header to Foundry.

Temporarily, copy the **TEST VALUE** also to the **DEFAULT VALUE** field. We will remove these credentials from there when the app comes to that point.

Note: Don't forget to review and clean up your Foundry project and its default credentials before publishing it in production!

Add the following **Response Output** parameters here (leave the other columns empty):

NAME	PATH	SCOPE	DATA TYPE	COLLECTION ID	FORMAT
responseList	$.[*]	response	collection		None
blogLocation	blogLocation	response	string	responseList	None
UNID	UNID	response	string	responseList	None
blogComment	blogComment	response	string	responseList	None
blogDate	blogDate	response	string	responseList	None
blogGroupId	blogGroupId	response	string	responseList	None

Your **Response Output** should now look like this:

NAME	PATH	SCOPE	DATA TYPE	COLLECTION I...	RECORD ID	FOR...
responseList	$.[*]	response	collection			None
blogLocation	blogLocation	response	string	responseList		None
UNID	UNID	response	string	responseList		None
blogComment	blogComment	response	string	responseList		None
blogDate	blogDate	response	string	responseList		None
blogGroupId	blogGroupId	response	string	responseList		None

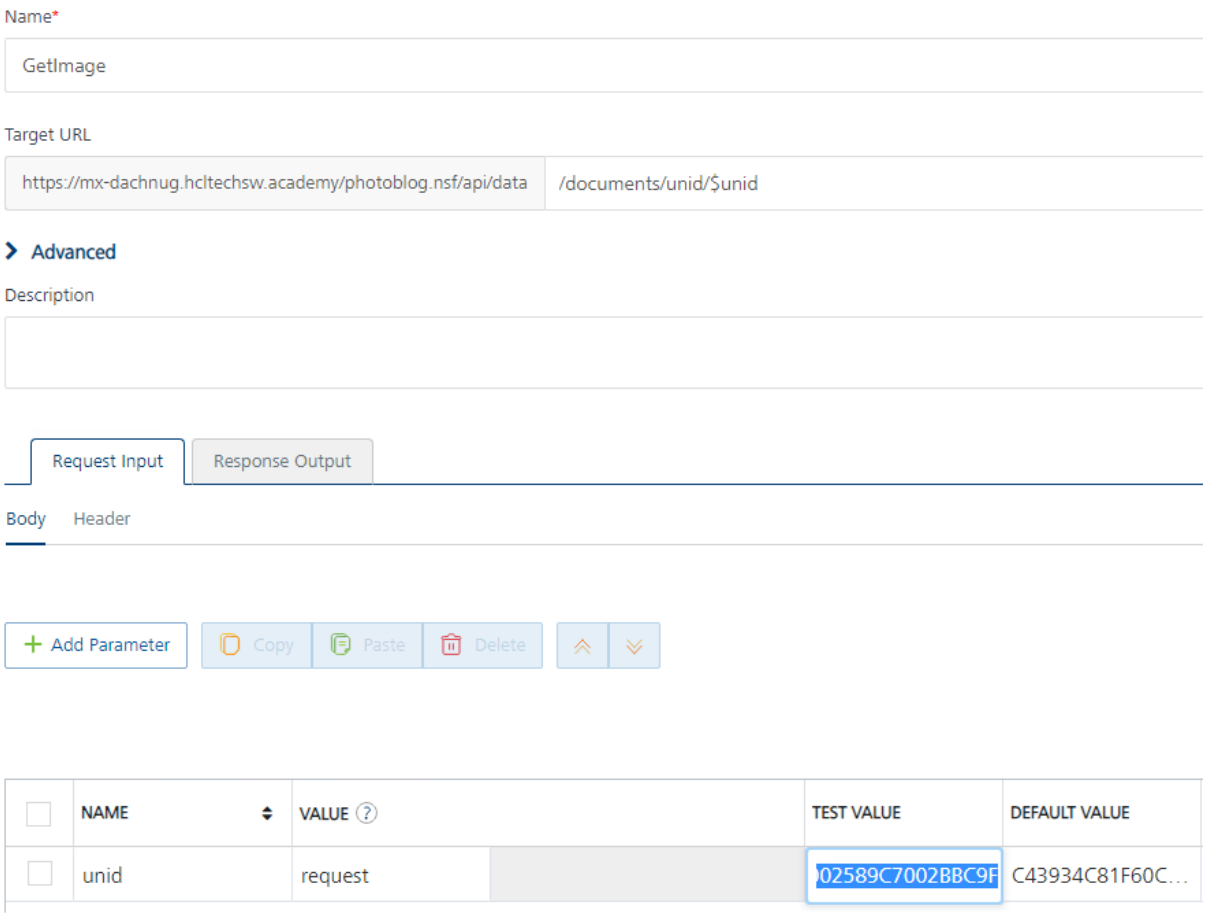
Click on **SAVE AND FETCH RESPONSE** to validate the output.

To retrieve the image of a location, we need a second service, because, RichText information in Domino can not be displayed and exposed via a view.

Create a second operation for this service, named **GetImage**.

It should retrieve data from an URL having the following suffix: **/documents/unid/$unid**

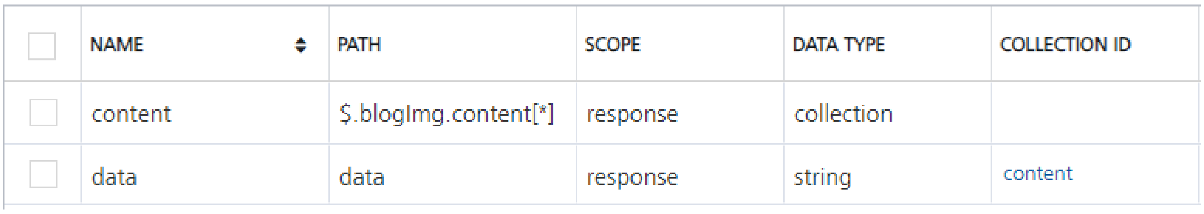
Add the **Authorization** header here, as you did before, plus the request parameter **unid** with a **TEST / DEFAULT VALUE** of **C43934C81F60CDBE002589C7002BBC9F.**



The **Response Output** contains 2 parameters:

NAME	PATH	SCOPE	DATA TYPE	COLLECTION ID	FORMAT
content	$.blogImg.content[*]	response	collection		None
data	data	response	string	content	None

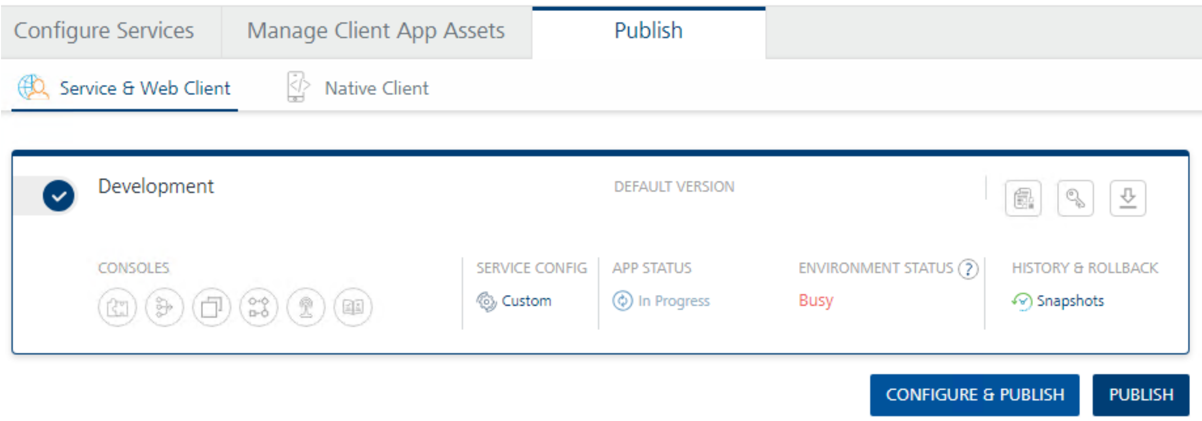
Your **Response Output** should now look like this:



Click **SAVE AND FETCH RESPONSE**, to see the results.

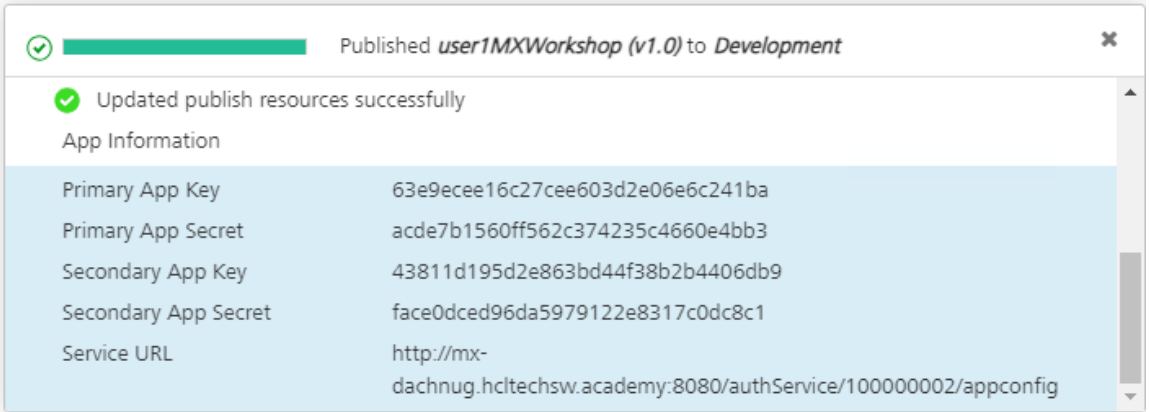
In a last step, we need to publish the service to the Foundry runtime environment, to make it consumable from the mobile app that we will create later.

Click on **Publish**.



You will see your accessible environments here. Enable your environment of choice and click **PUBLISH**.

If publishing succeeded, you will see the results.



Congratulations!

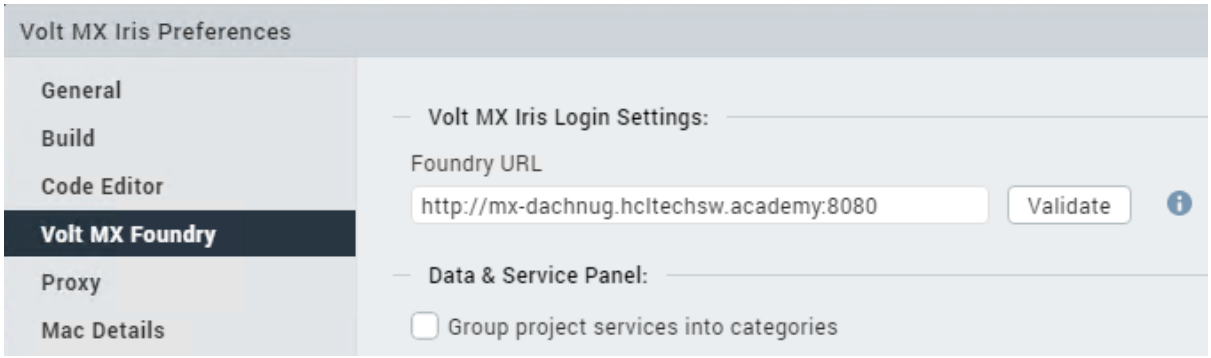
You have just finished creating the backend component for our HCL Volt MX App.

Building the Iris app

On your local machine, start the HCL Volt MX Iris client and login using the provided credentials.

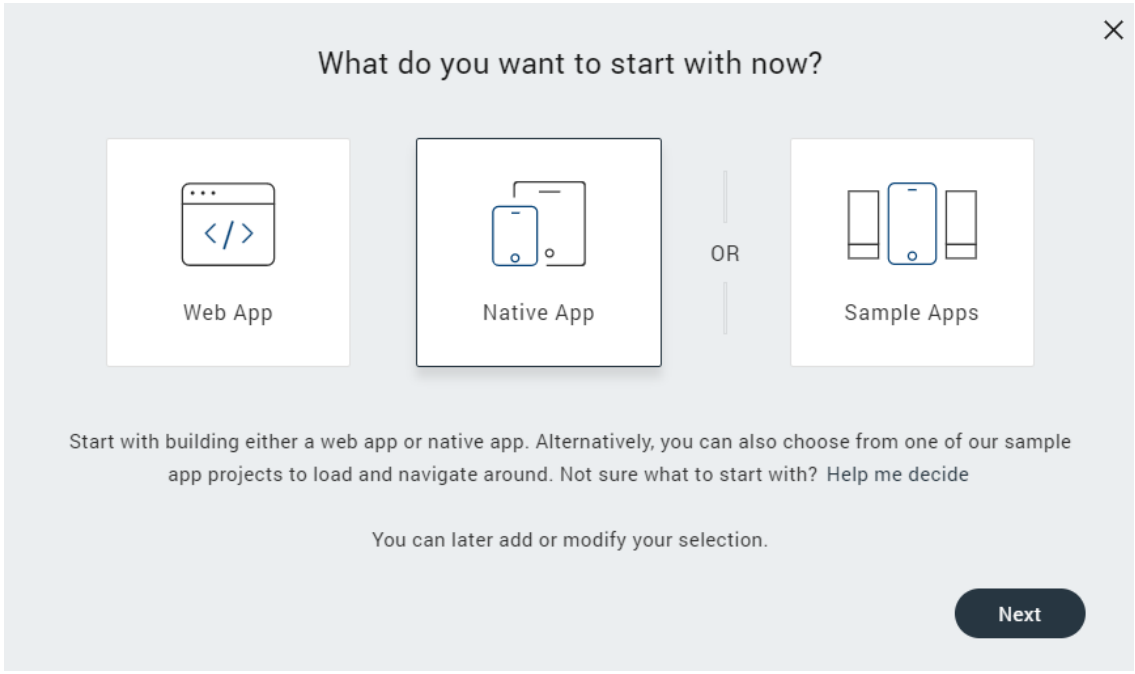
Click on **Edit > Preferences** and select **Volt MX Foundry** from the navigator.

Make sure that, your correct **Foundry URL** is defined here (the URL from screenshot below is just an example).



Close the Iris settings.

From the **Project** menu, select **New Project**.

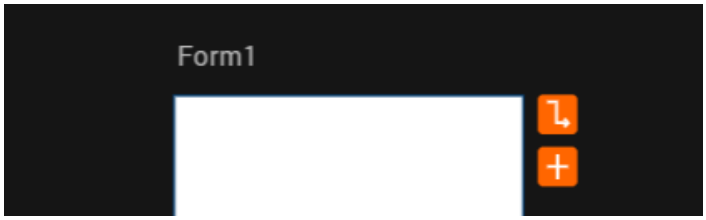


We want to create a new **Native App**, so select the middle option. Click **Next**.

Name the project **<userX>MXWorkshop**, which is the same name chosen for the Foundry part.

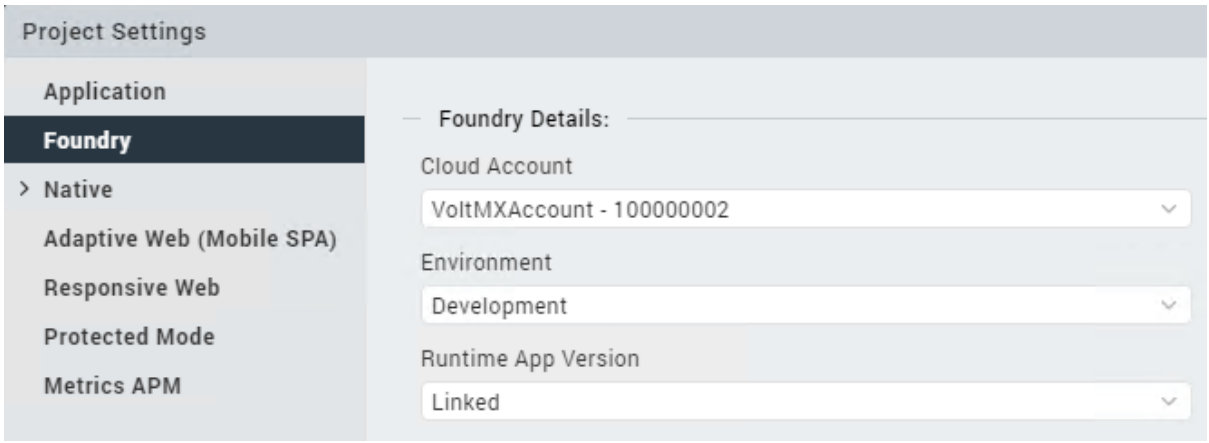
Click **Create**.

This will open the **STORYBOARD** perspective with an empty default form.



Navigate to **Project > Settings > Foundry**.

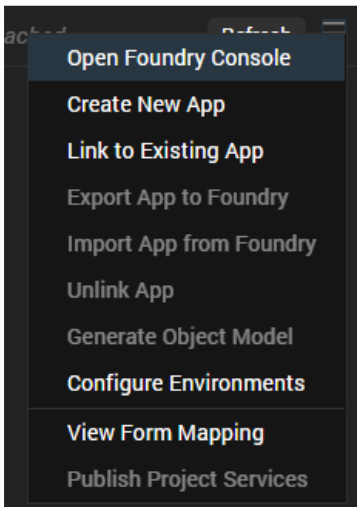
Select your correct Foundry cloud publishing environment from the **Environment** drop-down box.



Click **Done**.

Notice the right-hand **DATA & SERVICES** navigator. Here we want to see our set of services, which should be consumed by our app.

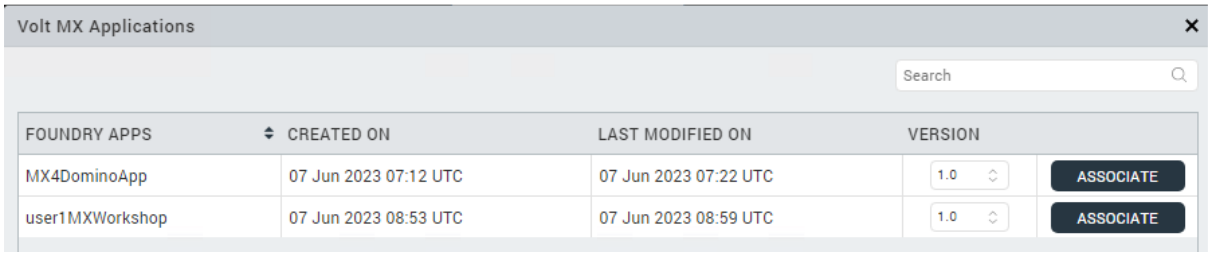
Click on the menu icon next to Refresh.



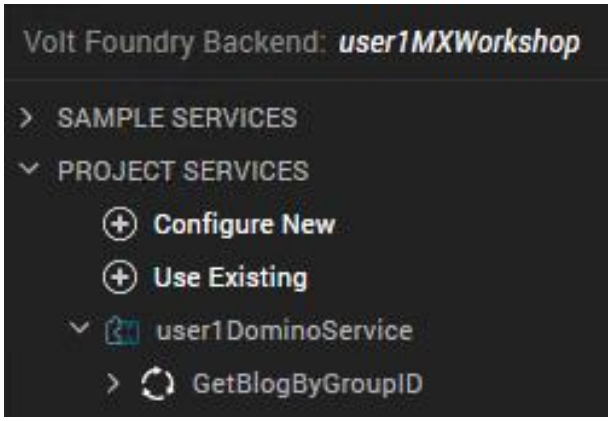
From the drop-down list, select **Link to Existing App**.

This should open a list of existing Foundry applications. If you are connected to the corrected Foundry backend, you should find your **<userX>MXWorkshop** project.

Click **ASSOCIATE** to connect it.



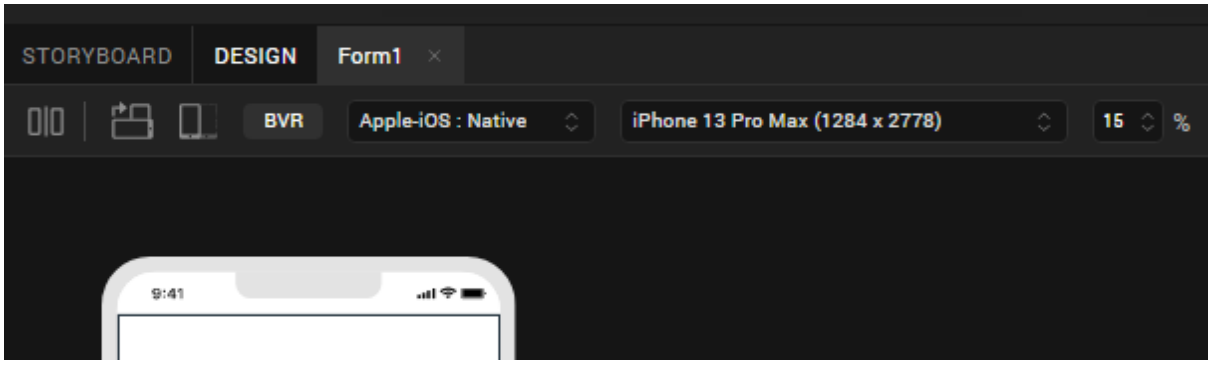
If you expand **PROJECT SERVICES** now, you should see the service we created earlier.



Now, let's create our first app page.

From the **PROJECT SERVICES** navigator, expand **<userX>DominoService**.

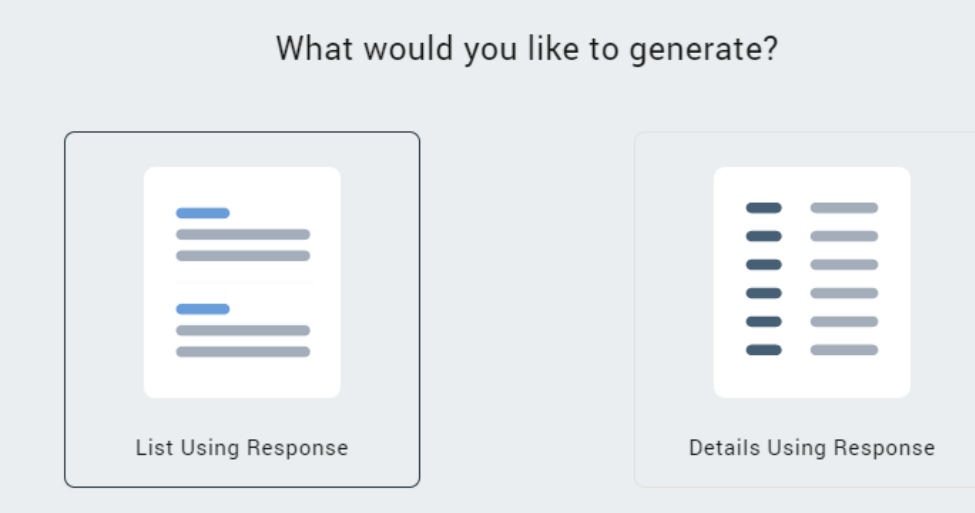
Double-click on **Form1** to open it in **DESIGN** mode.



Drag & Drop the **GetBlogByGroupID** operation to your **Form1**.

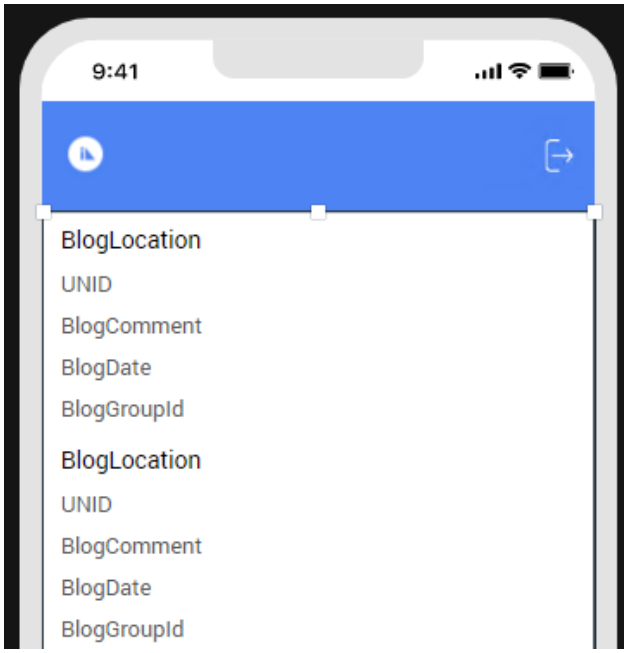
A dialog will open, asking for the desired UI representation for the service call's response.

Choose **List Using Response**.



Click **OK**.

Iris renders a base UI that connects to the service and show its default response.

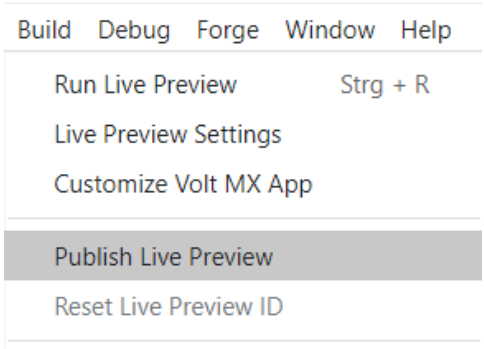


At this point, we can run our first test of the app on the physical device. You can either do that by connecting your iPhone or Android device to your Iris desktop or by connecting your device to a life preview hosted in Foundry.

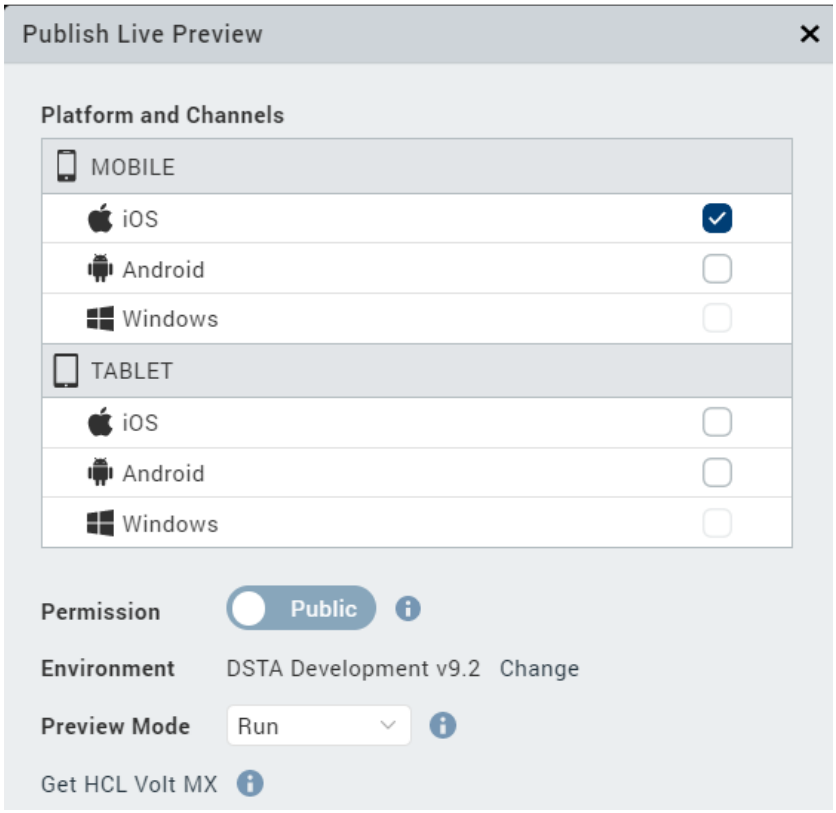
Option 1: Publish a Preview to Foundry

This is the easiest way to follow if you are running the HCL Volt MX Trial, as this comes with an environment for publication.

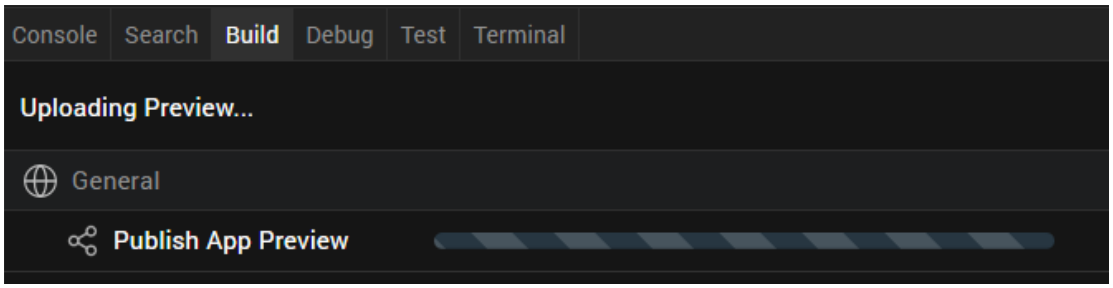
Select **Build > Publish Live Preview**.



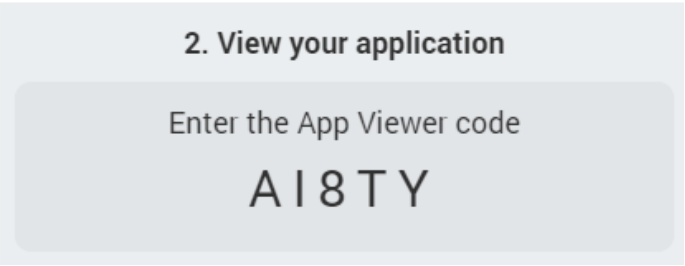
Select your device platform and keep the other settings.



Click **Publish**. This will generate and publish the app code.



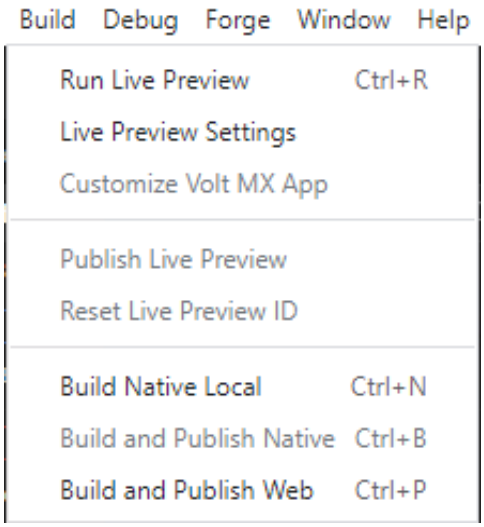
As the result, you will receive an App Viewer Code, to open the preview from your device.



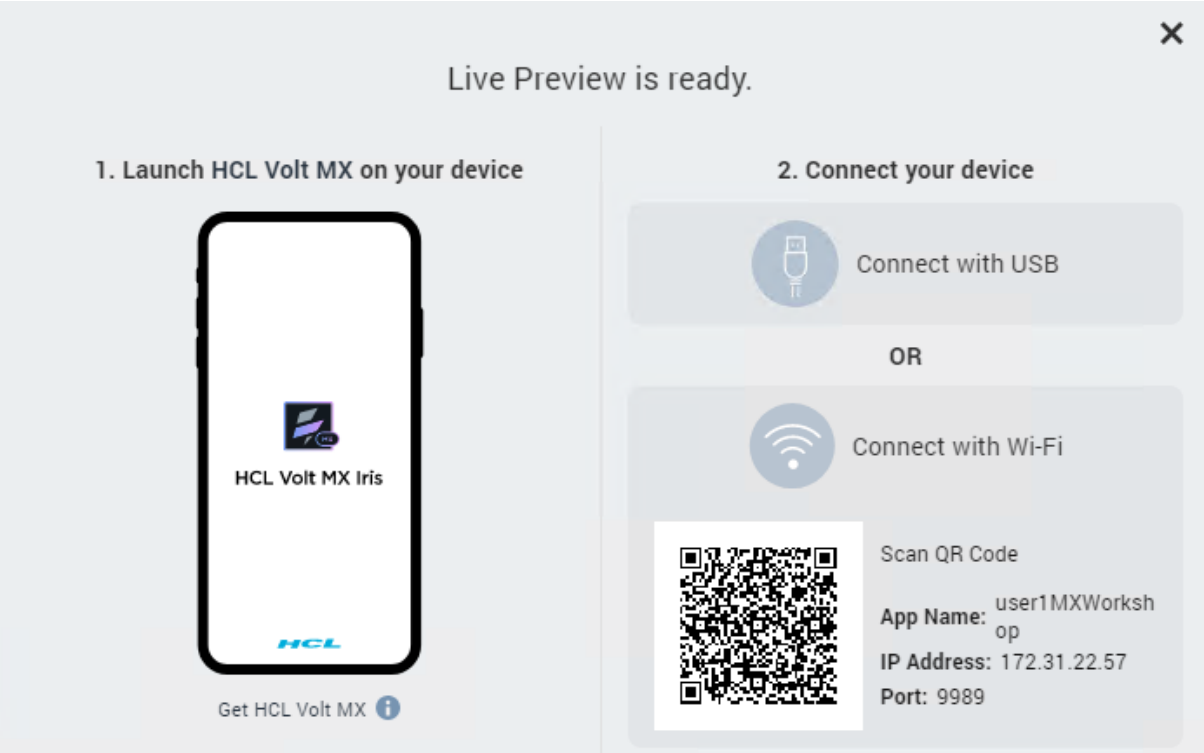
Option 2: Run Live Preview

If your device can connect to your Iris machine directly (bypassing firewalls and within a common network, e.g. WiFi), you can use this approach.

Select **Build > Run Live Preview**.



As the result, you will receive the information needed to open the preview from your device.

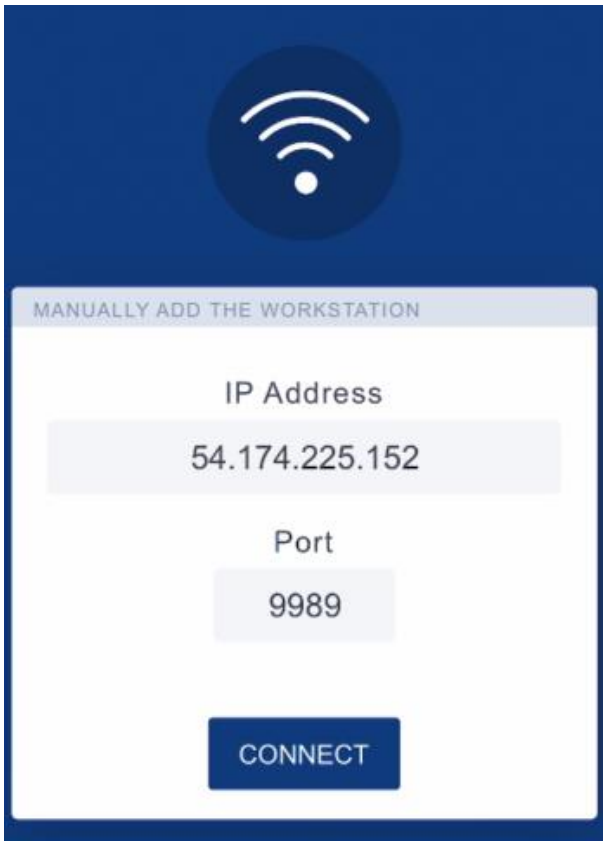


On your device, open the **Volt MX** app.

On first use, you will need to authenticate by entering your Volt MX Cloud credentials.

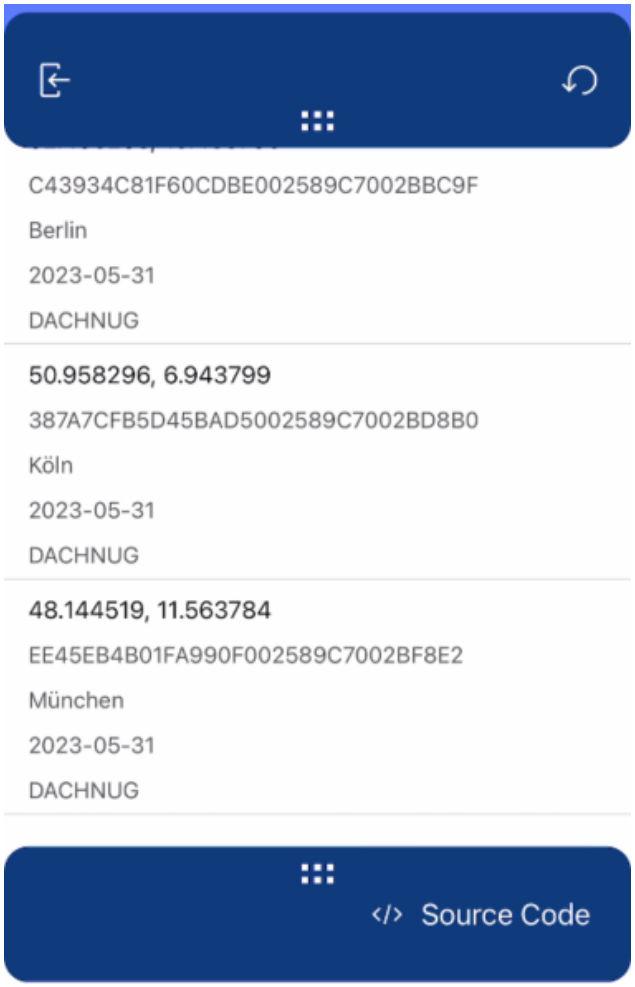
For *Option 1* (e.g. via Trial), use the **Cloud** Code shown in Iris and run the application.

For *Option 2*, select **Wi-Fi**. Enter the Public IP Address of your Iris VM, leave the default port 9989 and click **CONNECT**.



On a next screen, you will see the main navigation patterns that might, as an example, allow you to exit the app.

Click the blue **X** to leave the initial screen. This should open the app and, after a second, populate the list from the Domino service, based on the **DEFAULT** parameters we defined in the Foundry application.



Note: In case you notice any errors, investigate the following:

If you receive an SDK error, validate the Iris **Project Settings** and the **Foundry Environment** configured there.

In case you need to make changes here:

- From the **DATA & SERVICES** menu, select **Unlink App**.
- Select **Link to Existing App** and associate the **<userX>MXWorkshop** Foundry project again.
- **Refresh** the Foundry backend from the **DATA & SERVICES** panel.
- **Refresh** the project from the **Project** menu.
- Rebuild the Preview from Iris.

If you receive a Data Fetch error, this is most likely because of the missing or wrong authorization header for the service call. The easiest way to fix is to temporarily define a **DEFAULT VALUE** in the **Request Header Input** in your Foundry project. Then, re-publish the Foundry project.

Next, we will need to define some global variables within our app. These variables will allow us to modify, filter, pass and combine values passed from or to our services.

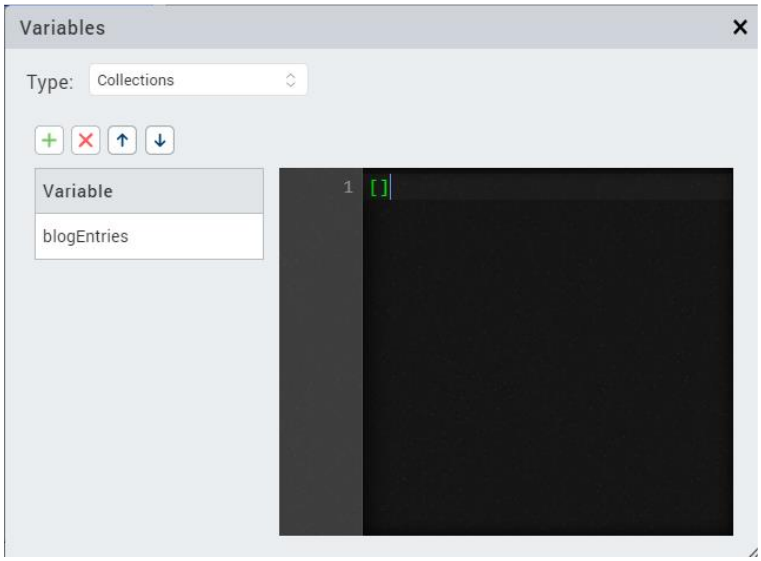
In Iris, navigate to **Edit > Global Variables** and create the following variables:

Simple (all with default “”):

- userName
- userCredentials

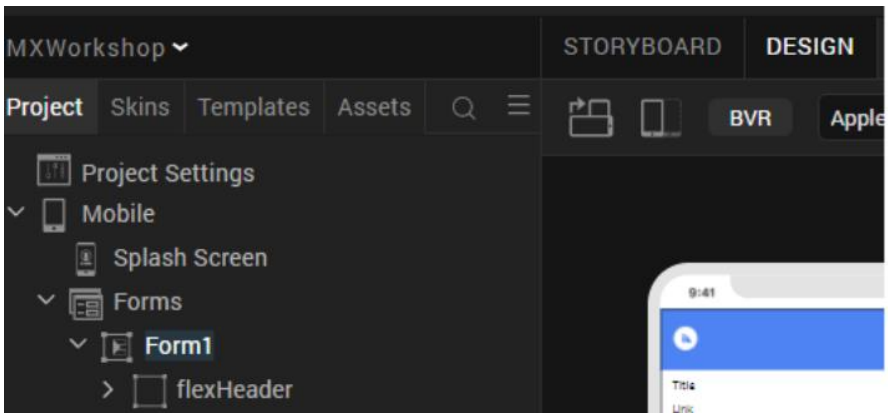
Collections (all with default []):

- blogEntries

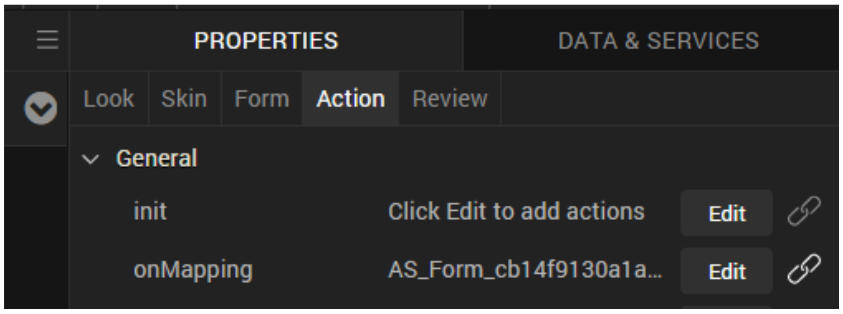


Close the **VARIABLES** dialog. Navigate to the **DESIGN** view in Iris and open **Form1**.

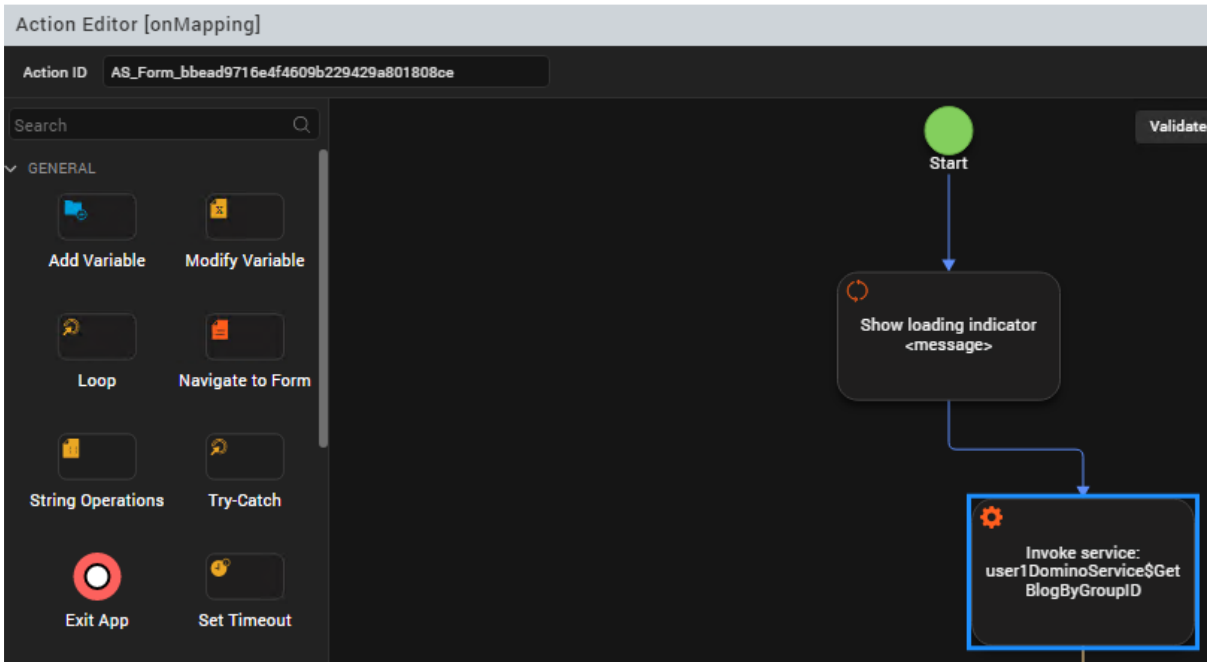
From the left-hand navigator, this is accessible via **Mobile > Forms > Form1**.



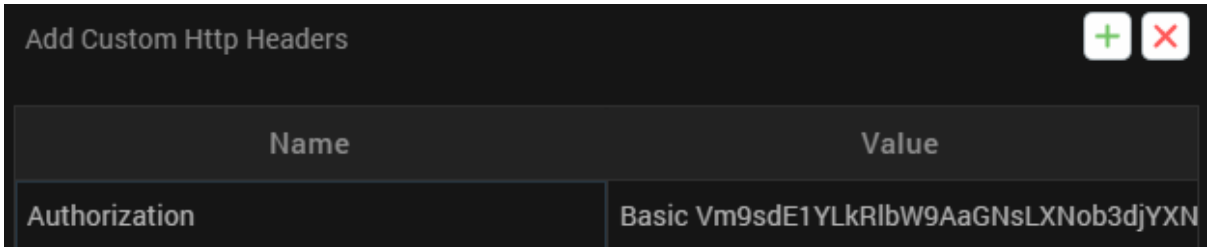
With focus on **Form1**, switch from **DATA & SERVICES** to **PROPERTIES** in the right-hand navigator. Click on **Action** there.



Find the **onMapping** event and click **Edit**. This will show the current sequence of operations created automatically during your drag & drop event.



Click on the **Invoke service: <userX>DominoService** module and add the **Authorization** header here.

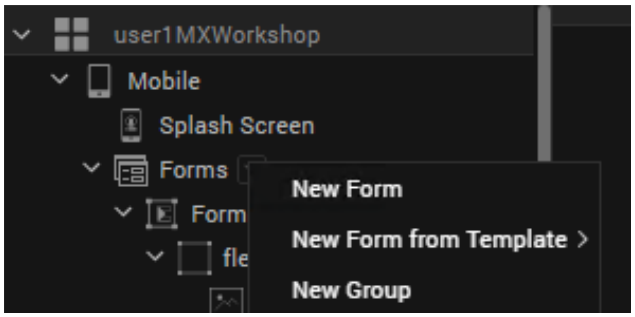


Note: Remember that, you have specified a **DEFAULT VALUE** for your service operations authorization in Foundry. From this point in our lab, you could remove that **DEFAULT VALUE**, as the app will pass this header to Foundry.

Click **Save**.

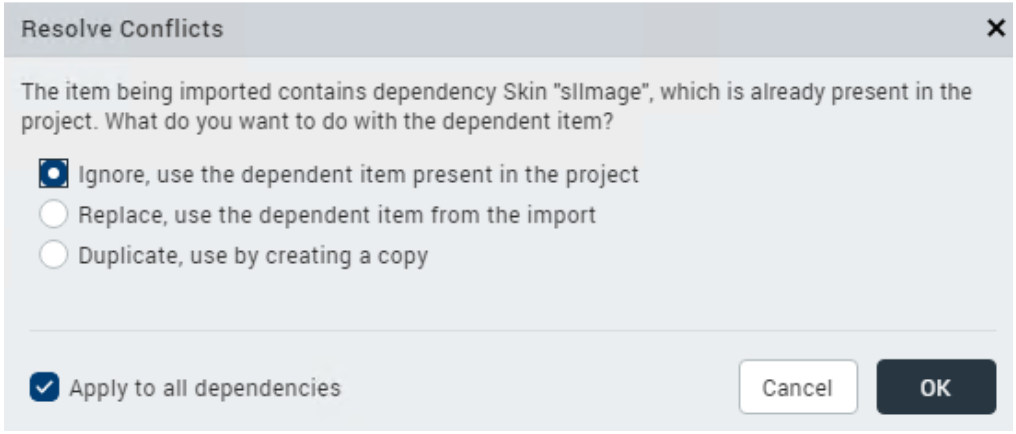
In our next step, we will implement a login form that will authenticate our user.

In Iris, right-click on **Mobile > Forms** and select **New Form**.

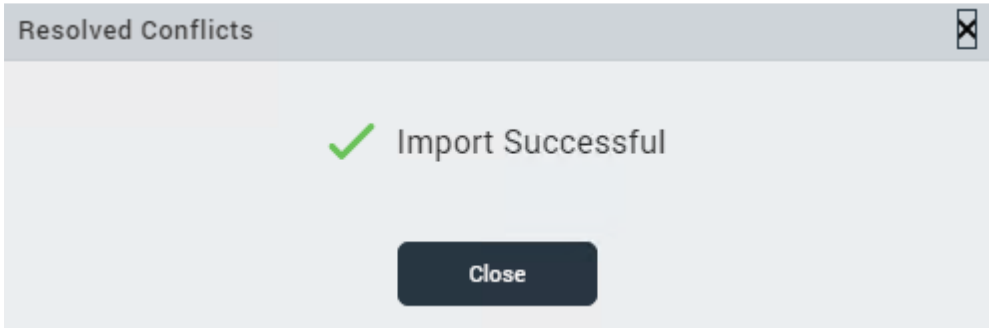


The left-hand navigator contains pre-built templates for different use cases, so scroll down until you reach the **Login** section.

Drag & drop the **konylogin.PinnacleLogin** part to your new form **Form2**.

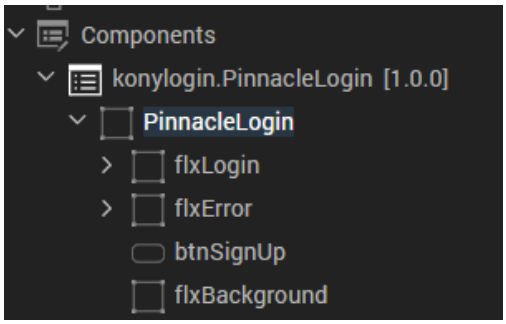


If a conflict is shown, select the **Ignore** option for all dependencies.



Right-click on the **PinnacleLogin** part and select **Edit Component**.

Your context root will change to **Components**.



Click on the **Dont have an account? Sign Up message** and delete that **btnSignUp** button.

Switch back to **Form2**. Notice that, your deletion is reflected here.

Click on the **Sign In** button and **Edit** the **onClick** Action here.

Add an **Add Snippet** component. This kind of component can take any JavaScript code.

Our code should do the following:

- Take the credentials entered on the login page
- Build a Base64-encoded authentication pattern
- Authenticate with the Domino service using that pattern

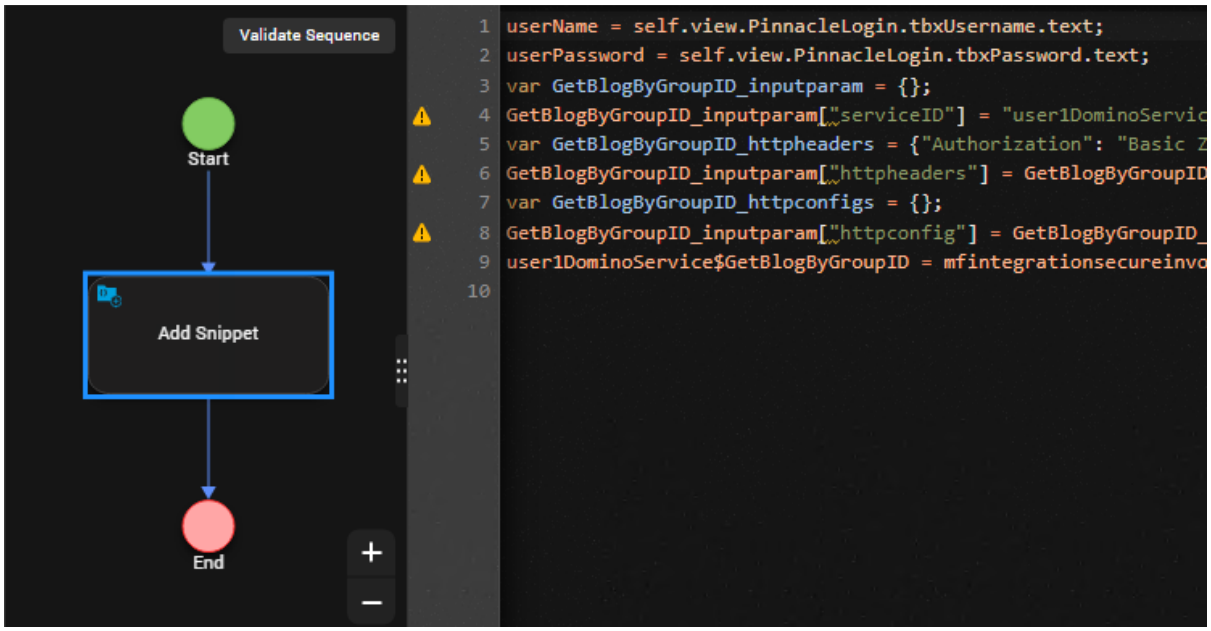
We will accomplish that step by step.

First, enter the following code here (**Note**: If you have a look into the **onMapping** code from **Form1**, you'll see that this is exactly what it does):

```
userName = self.view.PinnacleLogin.tbxUsername.text;
userPassword = self.view.PinnacleLogin.tbxPassword.text;
var GetBlogByGroupID_inputparam = {};
GetBlogByGroupID_inputparam["serviceID"] = "userXDominoService$GetBlogByGroupID";
var GetBlogByGroupID_httpheaders = {"Authorization": "Basic ZHVzZXI6Vm9sdEAxMjM0"};
GetBlogByGroupID_inputparam["httpheaders"] = GetBlogByGroupID_httpheaders;
var GetBlogByGroupID_httpconfigs = {};
GetBlogByGroupID_inputparam["httpconfig"] = GetBlogByGroupID_httpconfigs;
userXDominoService$GetBlogByGroupID =
mfintegrationsecureinvokerasync(GetBlogByGroupID_inputparam, "userXDominoService",
"GetBlogByGroupID", INVOKE_SERVICE_f120f7dc841d4adea8ae0141b24cec06_Callback);
```

Replace the 3 occurences of **<userX>** by your user ID here.

For **"Authorization",** use your credentials as you did before in Foundry. We will change that later.



Notice the last call. It invokes a callback service to handle the data retrieved. The ID looks cryptical, as this code can be merged by the developer from some **Add Service** modules that will generate this ID. It needs to be unique within your project. So for now, you can just copy that function name.

```
"GetBlogByGroupID", INVOKE_SERVICE_f120f7dc841d4adea8ae0141b24cec06_Callback);
```

Now, we need to add that missing function.

Insert the following code before the row starting with **userName**:

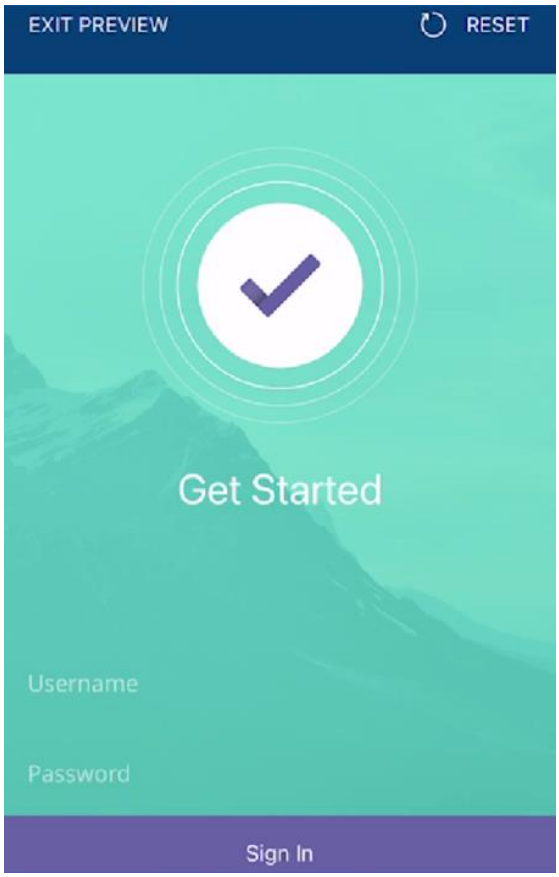
```
function INVOKE_SERVICE_f120f7dc841d4adea8ae0141b24cec06_Callback(status,
GetBlogByGroupID) {
  var httpResponse = GetBlogByGroupID.httpStatusCode;
  if (httpResponse != 200)  alert("Could not authenticate user");
  else
  {
    var ntf = new voltmx.mvc.Navigation("Form1");
    ntf.navigate();
  }
}
```

This function will forward the user to the initial form if authentication succeeded.

Click Save.

From the **STORYBOARD** view, right-click on **Form2** and select **Mark as Startup**.

Test the app.



If you enter some credentials (no matter what...), **Sign In** should take you to the Domino view.

As you might have noticed, the credentials entered here don't really matter. We still pass that **"Authorization"** parameter to the Domino services, which is today hard-coded.

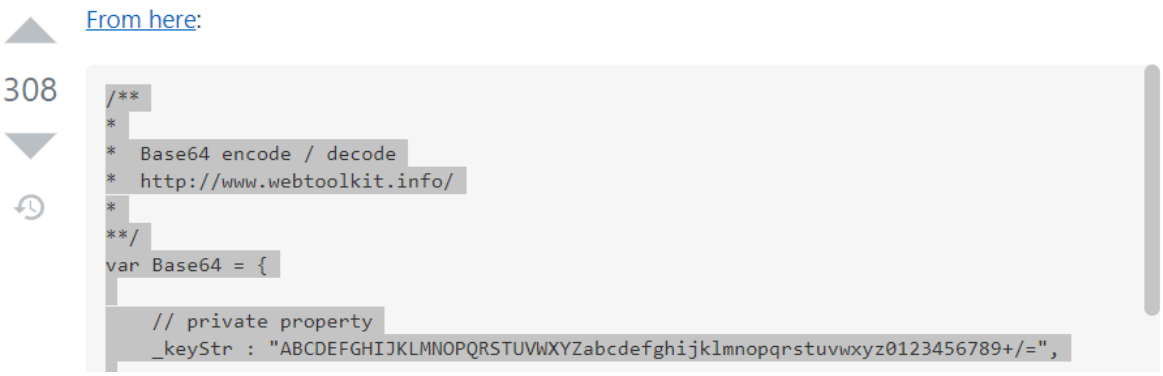
We will change that now.

First, we will need to find some way to run a **Base64** encoding on our set of credentials. You can use the **Base64** code pattern posted here:

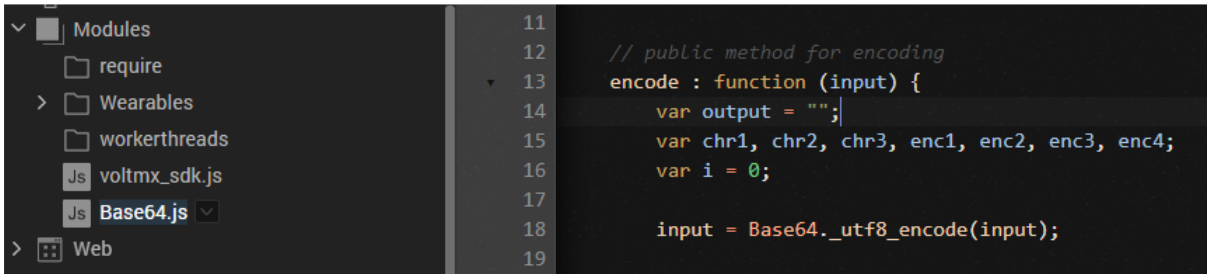
https://stackoverflow.com/questions/246801/how-can-you-encode-a-string-to-base64-in-javascript

There are other options available (but no **btoa / atob** function you might have in mind!)

In our example, copy the **Base64** code pattern.



In Iris, navigate to **Modules** and create a **New JS module** here. Copy the code to it.



Name the new module **Base64.js**.

From now, you can use that function from within your project.

Navigate back to your **Sign In** button code. Change the code as follows.

```
...
userName = self.view.PinnacleLogin.tbxUsername.text;
userCredentials =
Base64.encode(userName+":"+self.view.PinnacleLogin.tbxPassword.text);
var GetBlogByGroupID_inputparam = {};
GetBlogByGroupID_inputparam["serviceID"] = " userXDominoService $GetBlogByGroupID";
var GetBlogByGroupID_httpheaders = {"Authorization": "Basic "+userCredentials};
GetBlogByGroupID_inputparam["httpheaders"] = GetBlogByGroupID_httpheaders;
...
```

It should look similar to this:

```
  }
}

userName = self.view.PinnacleLogin.tbxUsername.text;
userCredentials = Base64.encode(userName+":"+self.view.PinnacleLogin.tbxPassword.text);
var GetBlogByGroupID_inputparam = {};
GetBlogByGroupID_inputparam["serviceID"] = "user1DominoService$GetBlogByGroupID";
var GetBlogByGroupID_httpheaders = {"Authorization": "Basic "+userCredentials};
GetBlogByGroupID_inputparam["httpheaders"] = GetBlogByGroupID_httpheaders;
var GetBlogByGroupID_httpconfigs = {};
GetBlogByGroupID_inputparam["httpconfig"] = GetBlogByGroupID_httpconfigs;
user1DominoService$GetBlogByGroupID = mfintegrationsecureinvokerasync(GetBlogByGroupID_inpu

```

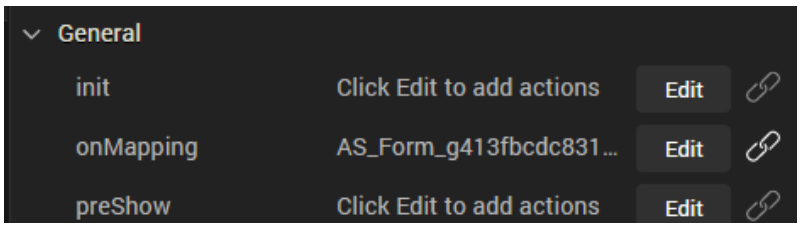
So, in this code we store the authentication credential of the current app user in the **userCredentials** global variable.

Preview your mobile app again.

From now, a **Sign In** takes the credentials entered here. You might double-check that by entering a wrong name or password here, which should result in an error message.

Let's now authenticate the Domino service by using these credentials instead of the defaults.

Navigate to the **Form1 onMapping Action**.

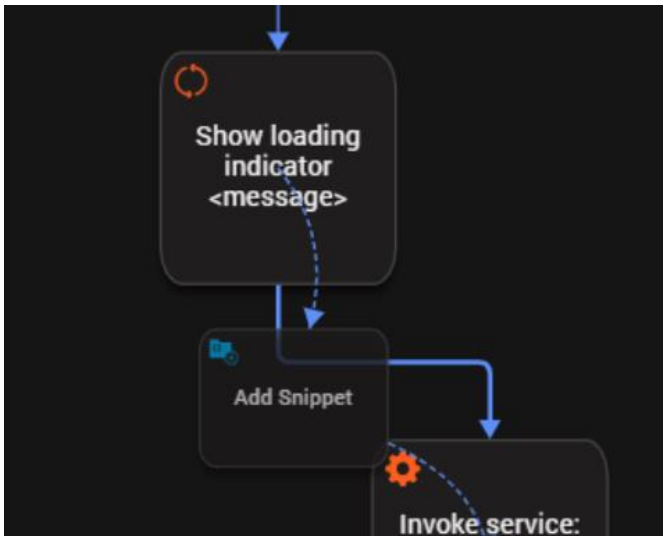


Click on the **Code View**. It shows the JavaScript code generated based on the current sequence of actions.

Copy everything between "**var self = this;**" and the closing "**}**" at the end of the code.

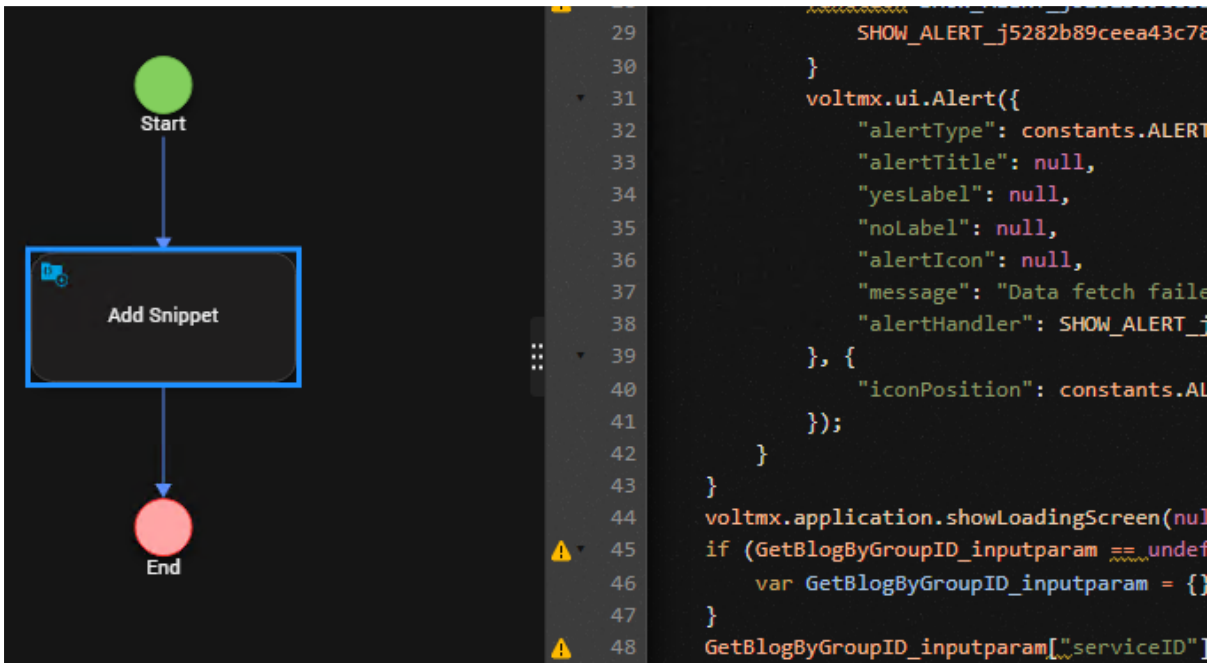
Switch back to the **Diagram View**.

Drag & drop a new **Add Snippet** and place it before the **Invoke_service** component.



Delete both the **Invoke_service** and the **Show loading indicator** component. This should delete everything except the **Add Snippet**.

Paste the copied code to your **Add Snippet**.



The **Add Snippet** now contains all code that was previously spread across several components. Now, we can modify this code.

Change the one line that builds the **httpheaders** set.

```
...
GetBlogByGroupID_inputparam["serviceID"] = "userXDominoService$GetBlogByGroupID";
    var GetBlogByGroupID_httpheaders = {"Authorization":"Basic "+userCredentials};
    GetBlogByGroupID_inputparam["httpheaders"] = GetBlogByGroupID_httpheaders;
    var GetBlogByGroupID_httpconfigs = {};
    GetBlogByGroupID_inputparam["httpconfig"] = GetBlogByGroupID_httpconfigs;
...
```

Note: We can access **userCredentials** from here, as this has been declared as a global variable.

Save the Action.

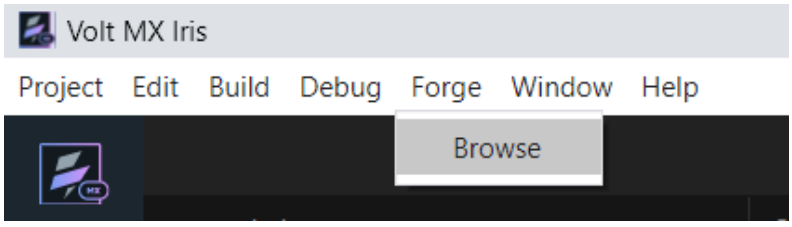
From that point, we won't need any authentication credentials in our Foundry app anymore. So, if not already done, you could delete both **TEST** and **DEFAULT VALUE** from your operations (and re-publish the Foundry app). Your app will send its individual user's credentials to Foundry instead.

Test your app, it should accept the correct credentials only and pass them from the authentication screen to the blog list.

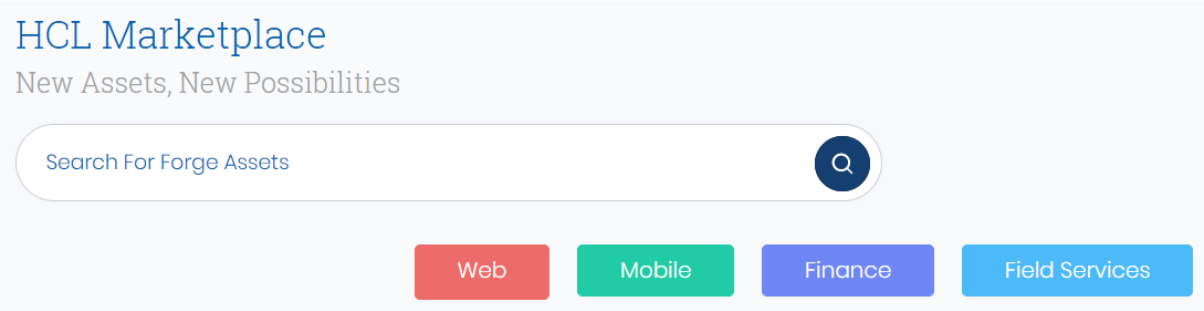
At this point, we will need some switch between our 2 app pages, especially in case you would like to add more forms. Because of the limited space on our mobile device, we would not implement a common navigator here. Instead, we want to make use of the so-called "Hamburger" menu: a menu icon that resides on top of the page and dynamically opens a navigator. It automatically hides after the user has selected some option from the menu.

We don't want to build this menu from scratch and, fortunately, Iris / Foundry comes with some pre-built asset for that.

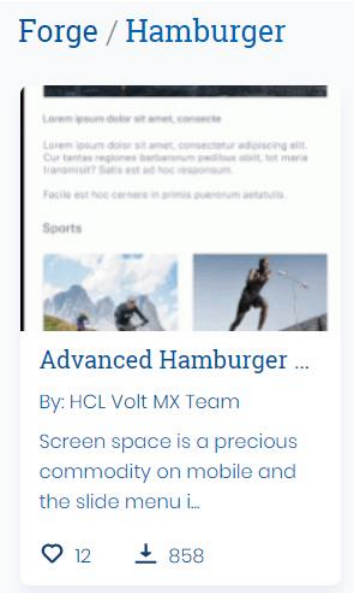
From your Iris client's main menu, select **Forge > Browse**. This will open the Foundry Marketplace.



Afterwards, you should see the following screen.

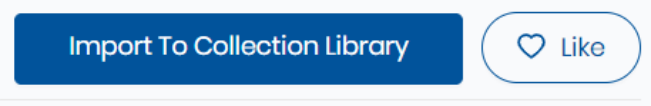


Search for “Hamburger”. It should return one result: The **Advanced Hamburger** menu.

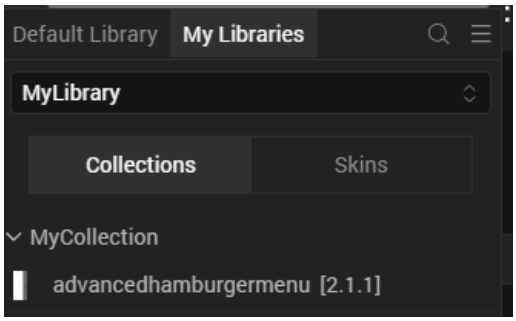


Open this result.

You are requested to login. If you have your Volt MX TrLogin, then switch to **version 2.1.1** and click on **Import To Collection Library** and create a local library named **MyLibrary** with **MyCollection** in it.

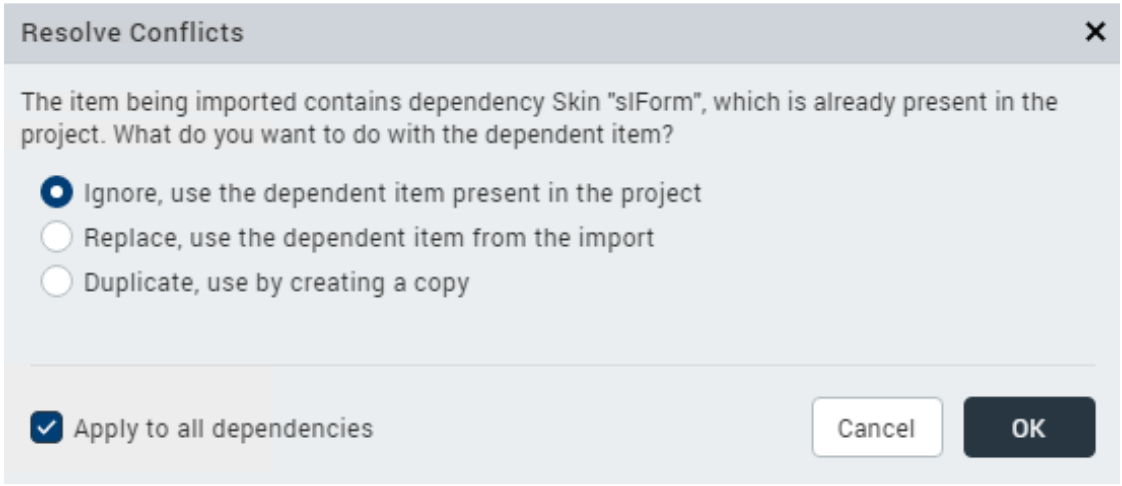


After import, you should see the downloaded component in your library.

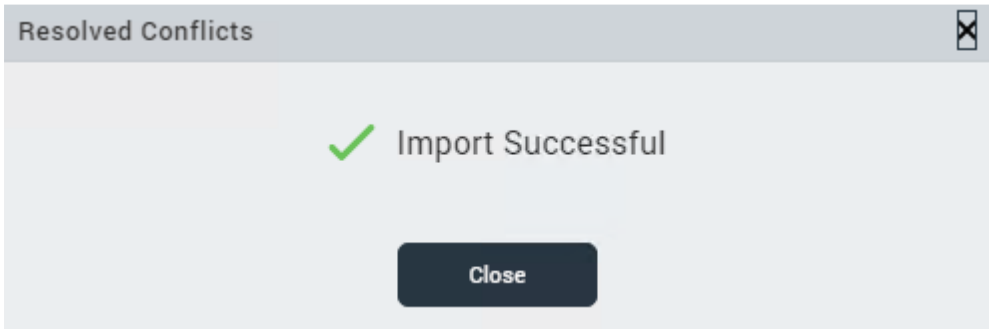


Switch to **Form1** in the **DESIGN** view.

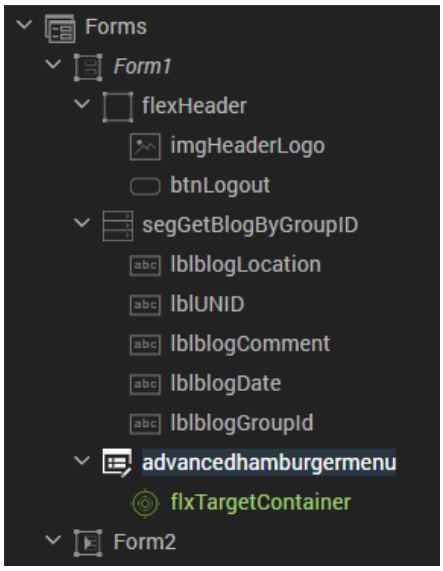
To add this **advancedhamburgermenu** to the project, drag & drop it to **Form1**.



If you are requested to decide on conflicts, select **Ignore**.



As you can see in the left-hand navigator, the page structure now looks like this for **Form1**:



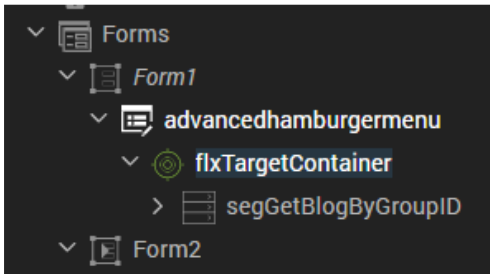
To leverage the new navigation feature, we need to make our existing form content part of the **flxTargetContainer**.

In **Form1**, right-click on **flexHeader** and select **Delete**.

Right-click on **segGetBlogByGroupID** and select **Cut**.

Right-click on **flxTargetContainer** and select **Paste**.

The result should look like this:



Due to the changed page structure, we need to adjust the code that refers to UI elements on **Form1**.

Open the **onMapping** code of that form. Find the following line:

```
self.view.segGetBlogByGroupID.setData(tempCollection5035);
```

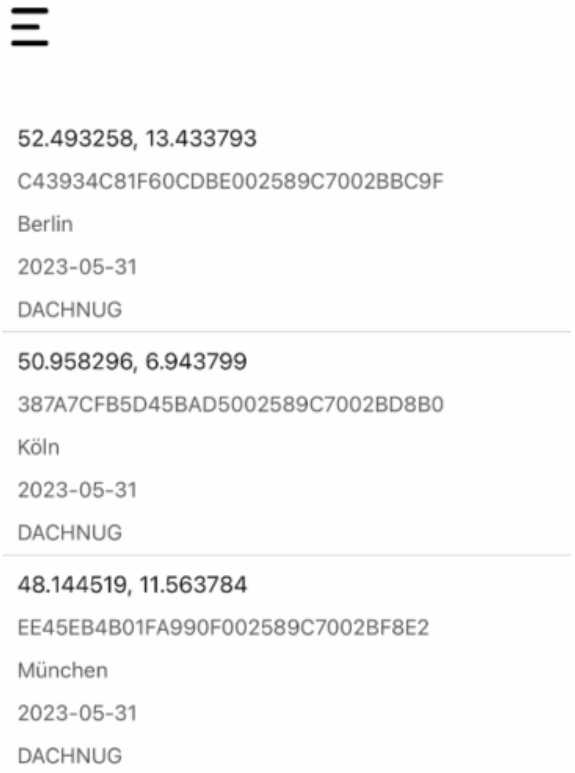
Change this code to:

```
self.view.advancedhamburgermenu.flxTargetContainer.segGetBlogByGroupID.setData(temp
Collection5035);
```

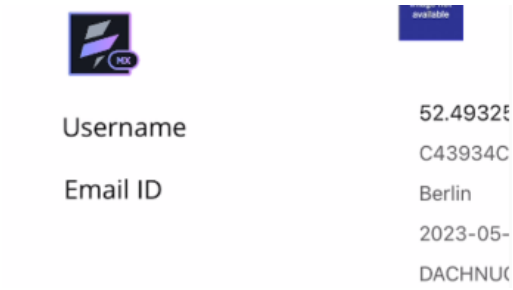
```
        });
    }
    self.view.advancedhamburgermenu.flxTargetContainer.segGetBlogByGroupID.setData(tempCollection5035);
} else {
    function SHOW_ALERT_cf2e6196d89d41c9a389be00fa788c82_Callback() {
```

Click **Save**.

Test the app. First, login and let it open the results page. To see any difference, flip the header menu (with **EXIT PREVIEW** and **RESET** on it) to the bottom of this page.



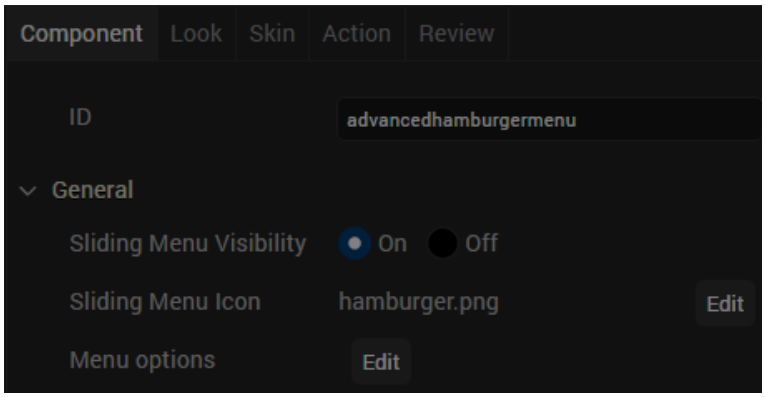
Click on the “Hamburger” menu.



As you can see, the menu comes with a set of placeholders like username, e-mail and pages to switch between. We will now configure those elements to fit our content.

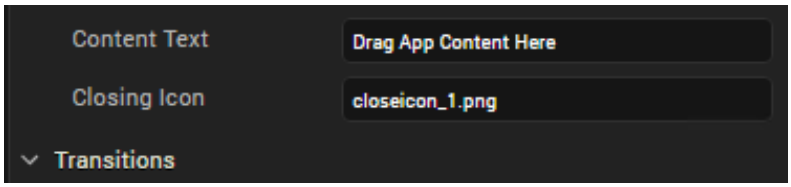
Navigate to **Form1 > advancedhamburgermenu**.

From right-hand navigator, switch to **PROPERTIES > Component** and click on **Menu options**.

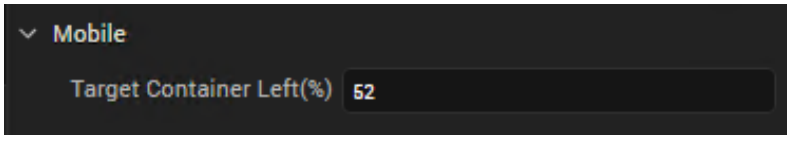


Delete all items by using the **DELETE** button.

Change the **Closing Icon** to **closeicon_1.png.**

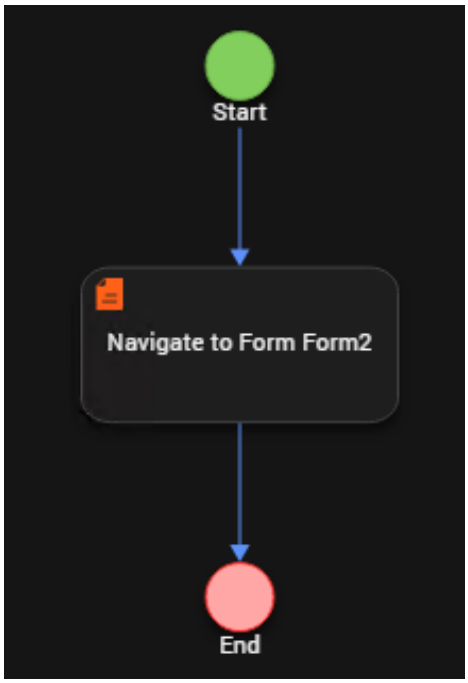


Set the **Mobile > Target Container Left(%)** option to **52**.



Navigate to **Actions.**

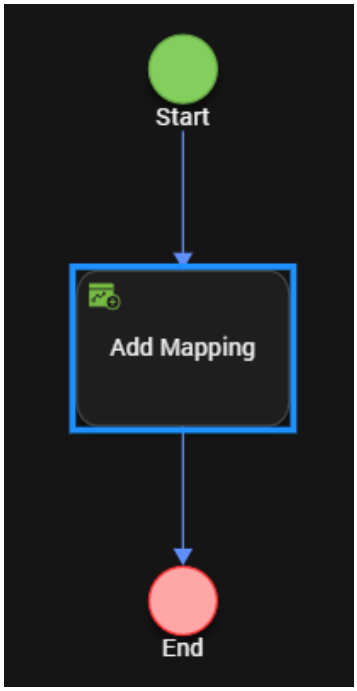
Find the **onFooterClick** Action and add an **Navigate to Form** action here that jumps to **Form2**.



Last, but not least, we want to show our user name in that menu.

Navigate to **Form1** and add an **init** Action there.

The action should contain a mapping, therefore add an **Add Mapping** module.



Map the following elements:

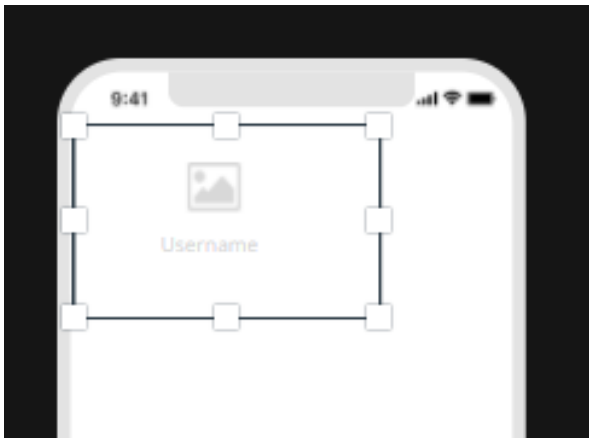
- **Variables > userName** to **Forms > Form1 > Component Instance... > headingText**

Again, it might be easier to just use **an Add Snippet** element here that just contains 1 line of code, as you can see in the **Code View**.

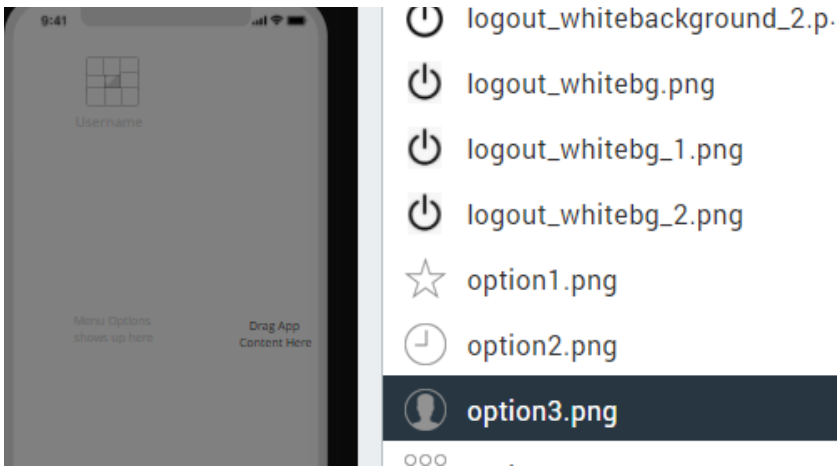
Click **Save**.

On **Form1**, click at the **advancedhamburgermenu** element and select **Edit Component**.

Remove the **Email ID** placeholder (or field **lblHeaderText2**). The component should look like this:

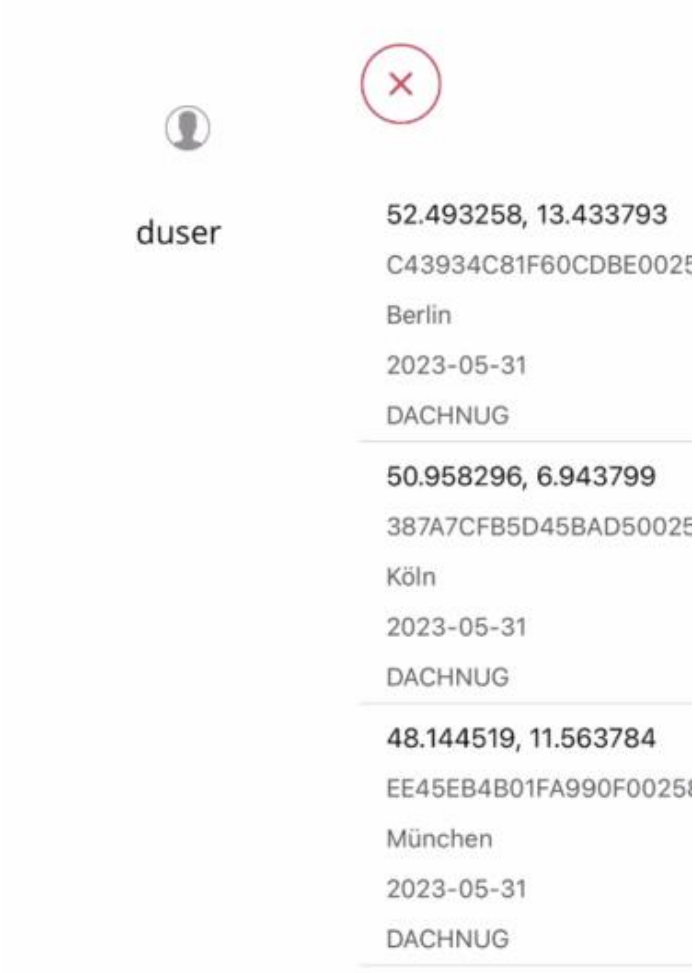


Change the image source to **option3.png**:



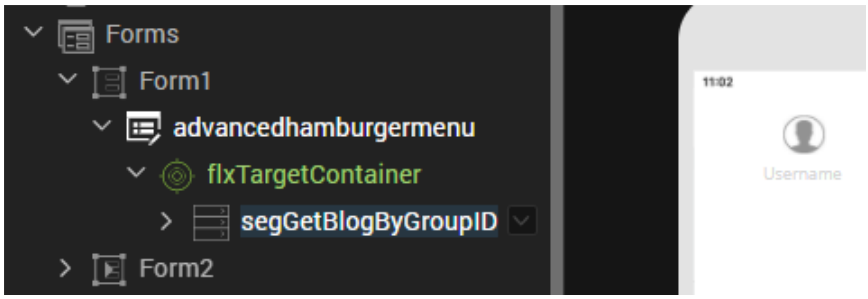
Finally, build and test your app. After authentication, it should now show data from user login in the Hamburger menu.

The **Log Out** option should also open the second form.

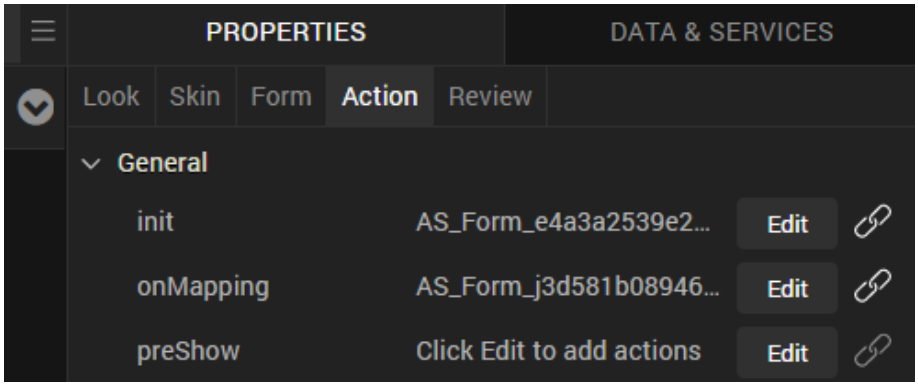


In our next step, we will visualize the information from our Domino database to provide a Map with recently visited locations instead of a list.

Expand the form's hierarchy and delete the **segGetBlogByGroupID** elements.



Remove the **onMapping** Action from **Form1** by clicking the **Unlink** icon next to **Edit**.



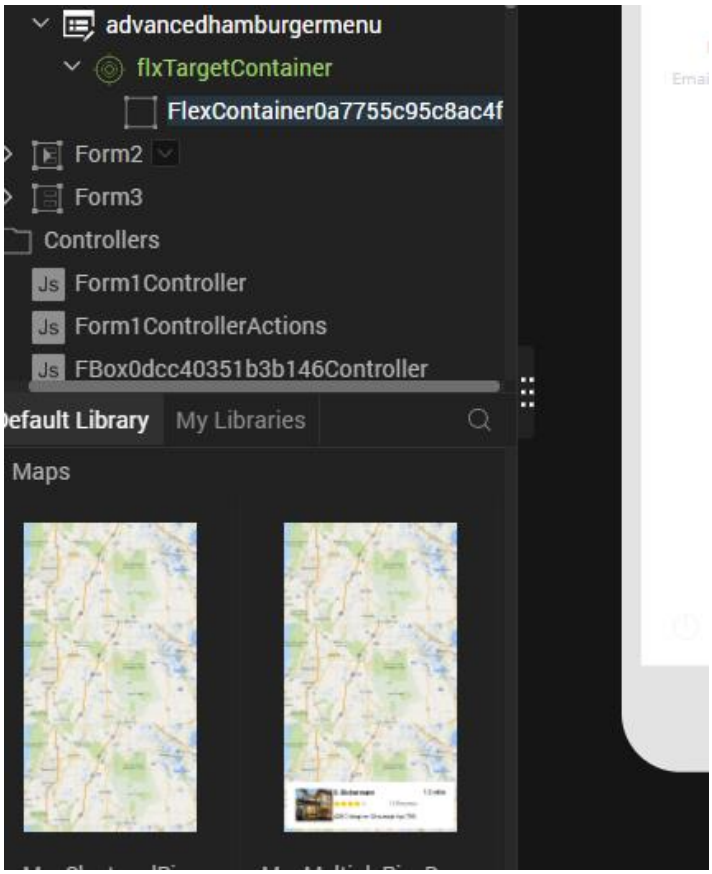
Confirm that, mapped actions will be deleted.

Highlight the **flxTargetContainer** element from the hierarchy.

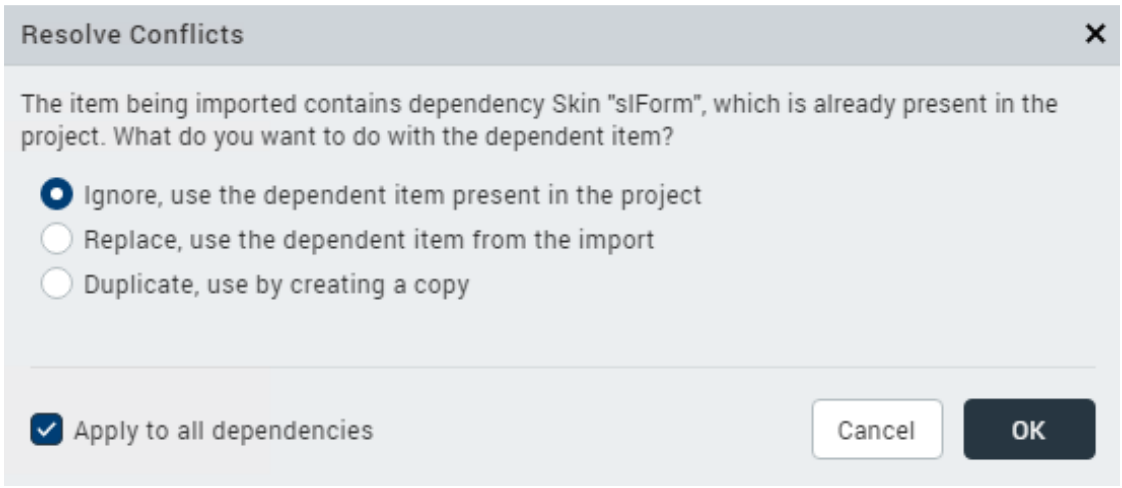
From left-hand navigator, click on **Default Library**.

Expand **Maps**.

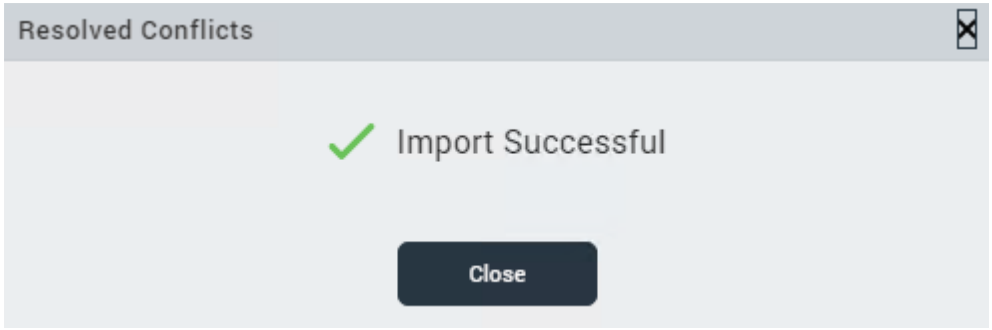
Drag the **MapMultiplePinsDockedCallout** element to the form.



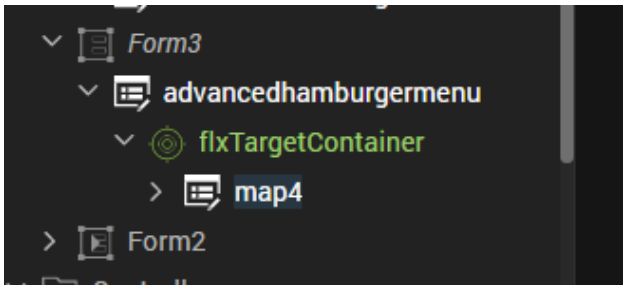
If you are requested to decide on conflicts, select **Ignore**.



Import should be successfull.

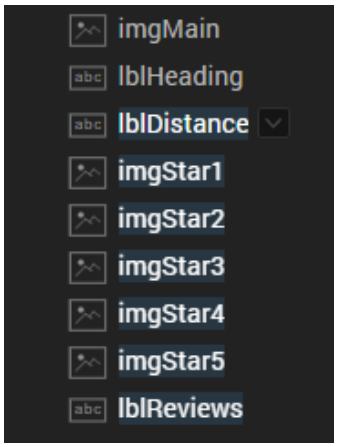


The new object should group under the **flxTargetContainer** element. If not, use **Cut** & **Paste** as you did before.

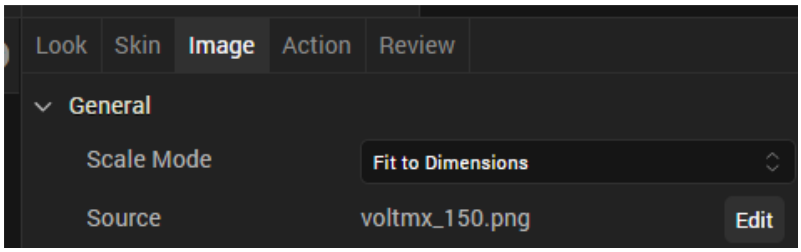


Right-click on **map4** and select **Edit Component**.

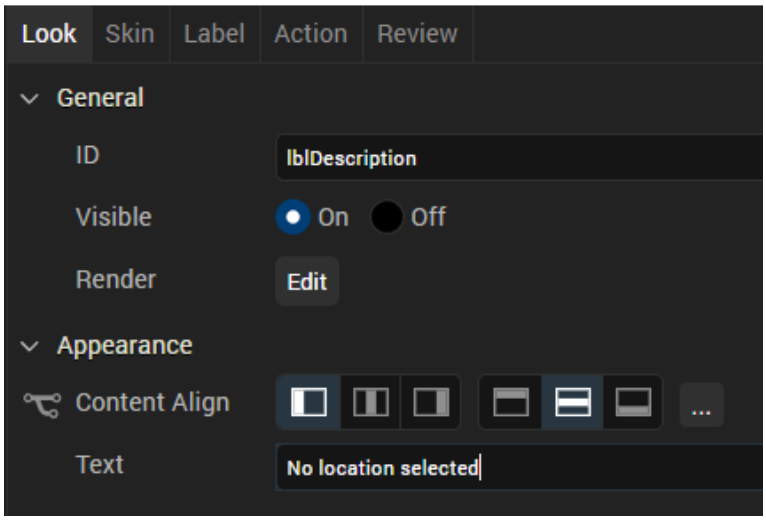
Expand **flxMain** and delete all children except **imgMain**, **lblHeading** and **lblDescription**.



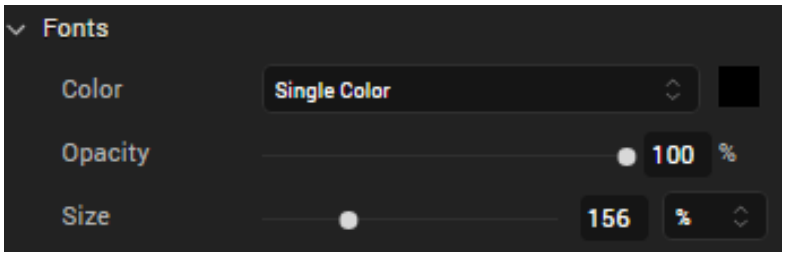
Click on **imgMain** and change its source to **voltmx_150.png**.



Set the default text for **lblHeading** and **lblDescription** to some default.



Increase the font size of **lblHeading** to **156%**.



Save the component.

From left-hand-navigator, scroll down and expand the **Modules** section from the **com.konymp.map4** component.

If a Map is getting rendered, the **map4Controller.js** initializes the location(s) shown as pins. By default, this is done based on some dummy data.

```
define(function() {

  return {
    addDataToMap : function(){
      this.previous ="1";
      this.locationData =[
       {
          id:"1",
          name : "pin1",
          lat:"40.748817",
          lon:"-73.985428",
          image : "kony_mp_map04_blue_pin.png",
          lblheading : "Bikermann Davis",
          lbldistance :"1.2 Miles",
          dec : "428 Cristopher Stravenue Apt. 495",
          showcallout: false
        },
```

We would like to set the pins based on our Domino application data.

Mark everything between **this.previous="1";** and **this.view.mapView.locationData = this.locationData;** and replace that part by the following code:

```
this.locationData = new Array();
for(var v =0; v< blogEntries.length; v++) {
       var curLocation =  {
              id: blogEntries[v][0],
              name: blogEntries[v][1],
              lat: blogEntries[v][2],
              lon: blogEntries[v][3],
              image: blogEntries[v][4],
              lblheading: blogEntries[v][1],
              lbldescription: blogEntries[v][5],
              showcallout: false
        };
        this.locationData.push(curLocation);
}
```

The result should look like this:

```
    addDataToMap : function(){
      this.previous ="1";
      this.locationData = new Array();

      for(var v =0; v< blogEntries.length; v++)
      {
        var curLocation =  {
            id : blogEntries[v][0],
            name : blogEntries[v][1],
            lat:blogEntries[v][2],
            lon:blogEntries[v][3],
            image : blogEntries[v][4],
            lblheading : blogEntries[v][1],
            lbldescription : blogEntries[v][5],
            showcallout: false
        };

        this.locationData.push(curLocation);
      }

      this.view.mapView.locationData = this.locationData;
```

Our Map should dynamically extract the locations to show from a global variable **blogEntries**, that we created before.

Scroll down to the **pinClicked** function and delete the **lblDistance** reference. Also, change the **lblDescription** source to **location.lbldescription**.

```
},
pinClicked : function(location){
  this.view.mapView.updatePins([{id:this.previous,image:"kony_mp_map04_red_pin.p
  this.previous=location.id;
  this.view.lblHeading.text = location.lblheading;
  this.view.lblDescription.text = location.lbldescription;
}
```

Retrieval of the blog entries should be part of the login process within the app. Remember...previously, it was part of the **Form1 onMapping** activity.

So, save and close the **map4Controller.js** for now.

Switch back to our **Form2**, which provides the login capabilities.

Open the **onClick** event of our login button.

Find the callback function that is responsible for navigating to our **Form1** after successful login.

In the **else** path of this function, add the following code:

```
…
else
{
        var tempData5973 = GetBlogByGroupID.responseList;
        blogEntries = new Array();
        var blogCount=0;
        for (var each4943 in tempData5973) {
          blogCount ++;
          var blogEntry = new Array();
          blogEntry.push(blogCount.toString());
          blogEntry.push(tempData5973[each4943]["blogComment"]);
          blogEntry.push(tempData5973[each4943]["blogLocation"].split(", ")[0]);
          blogEntry.push(tempData5973[each4943]["blogLocation"].split(", ")[1]);
          blogEntry.push(tempData5973[each4943]["UNID"]);
          blogEntry.push(tempData5973[each4943]["blogDate"]);
          blogEntries.push(blogEntry);
        }
        var ntf = new voltmx.mvc.Navigation("Form1");
        ntf.navigate();
}
```

```
var httpResponse = GetBlogByGroupID.httpStatusCode;
if (httpResponse != 200)  alert("Could not authenticate user");
else
{
    var tempData5973 = GetBlogByGroupID.responseList;
    blogEntries = new Array();
    var blogCount=0;
    for (var each4943 in tempData5973) {
      blogCount ++;
      var blogEntry = new Array();
      blogEntry.push(blogCount.toString());
      blogEntry.push(tempData5973[each4943]["blogComment"]);
      blogEntry.push(tempData5973[each4943]["blogLocation"].split(", ")[0]);
      blogEntry.push(tempData5973[each4943]["blogLocation"].split(", ")[1]);
      blogEntry.push("kony_mp_map04_blue_pin.png");
      blogEntry.push(tempData5973[each4943]["blogDate"]);
      blogEntries.push(blogEntry);
    }

    var ntf = new voltmx.mvc.Navigation("Form1");
    ntf.navigate();
}
}
```

This code takes the response from our Domino service and stores the results into a global variable for later use in the Map.

Save the action.

Save your open files and test the resulting application.

Note: Click on the pins, to see the related information from the location.

We are still missing the location images.

Navigate back to the **map4Controller.js** file.

Between the **addDataToMap** and **pinClicked** functions, add the following code:

```
getBlogImageCallback: function (status, GetImage) {
      voltmx.application.dismissLoadingScreen();
      if (GetImage.opstatus == 0) {
        var tempData5974 = GetImage.content[1].data;
        tempData5974 = tempData5974.substring(tempData5974.indexOf("<png>")+5);
        tempData5974 = tempData5974.substring(0, tempData5974.indexOf("</png>"));
        this.view.imgMain.base64=tempData5974;
      }
},

getBlogImage: function (unid) {
      var GetImage_inputparam = {};
      GetImage_inputparam["serviceID"] = "userXDominoService$GetImage";
      GetImage_inputparam["unid"] = unid;
      GetImage_inputparam["httpheaders"] = {"Authorization": "Basic
"+userCredentials};
      GetImage_inputparam["httpconfig"] = {};
      userXDominoService$GetImage =
mfintegrationsecureinvokerasync(GetImage_inputparam, "userXDominoService",
"GetImage", this.getBlogImageCallback);
},
```

The result should look like this:

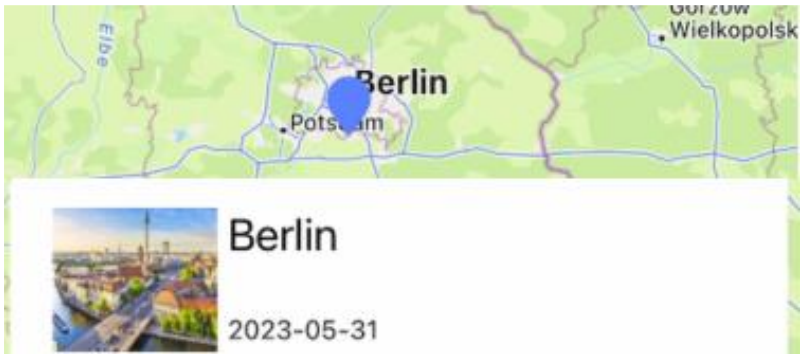
```
    this.view.mapView.locationData = this.locationData;
  },

  getBlogImageCallback: function (status, GetImage) {
    voltmx.application.dismissLoadingScreen();
    if (GetImage.opstatus == 0) {
      var tempData5974 = GetImage.content[1].data;
      tempData5974 = tempData5974.substring(tempData5974.indexOf("<png>")+5);
      tempData5974 = tempData5974.substring(0, tempData5974.indexOf("</png>"));
      this.view.imgMain.base64=tempData5974;
    }
  },

  getBlogImage: function (unid) {
    var GetImage_inputparam = {};
    GetImage_inputparam["serviceID"] = "user1DominoService$GetImage";
    GetImage_inputparam["unid"] = unid;
    GetImage_inputparam["httpheaders"] = {"Authorization": "Basic "+userCredentials};
    GetImage_inputparam["httpconfig"] = {};
    user1DominoService$GetImage = mfintegrationsecureinvokerasync(GetImage_inputparam, "user1D
  },

  pinClicked: function(location){
    this.view.mapView.updatePins( [ {id: this.previous, image: "kony_mp_map04_red_pin.png"},
```

Add the following lines to the **pinClicked** function:

```
this.getBlogImage(location.image);
```

```
    this.view.lblDescription.text = location.lbldescription;
    this.getBlogImage(location.image);
  },
```

If you now click on a pin from your map, the image stored in the Domino database is retrieved and displayed.



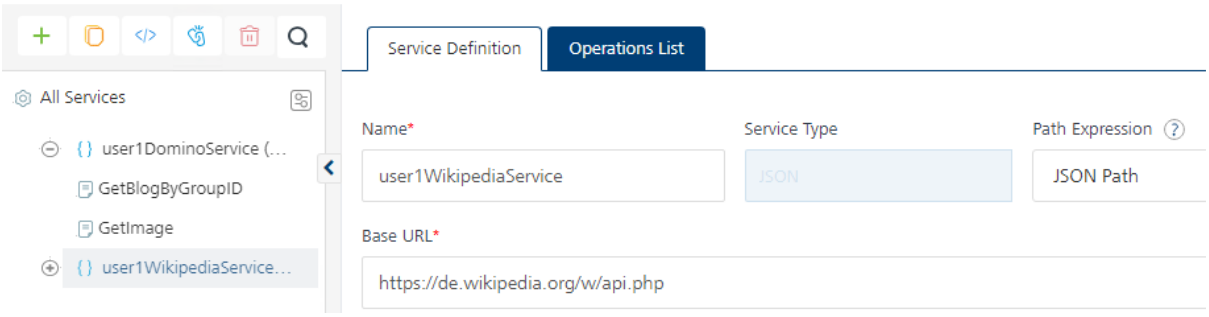
Optional: Add a Wikipedia service

With Volt MX, we can not only talk to a Domino backend. Foundry has pre-built connectors for almost any technology and data and could be even extended by custom connectors. For our second scenario, we will combine information that comes from Domino with a different backend: Wikipedia.

For doing that, complete the following steps, starting with Foundry:

Add a new service **userXWikipediaService** to your **userXMXWorkshop** Foundry project. It uses **JSON** to connect to the following **Base URL**:

https://de.wikipedia.org/w/api.php



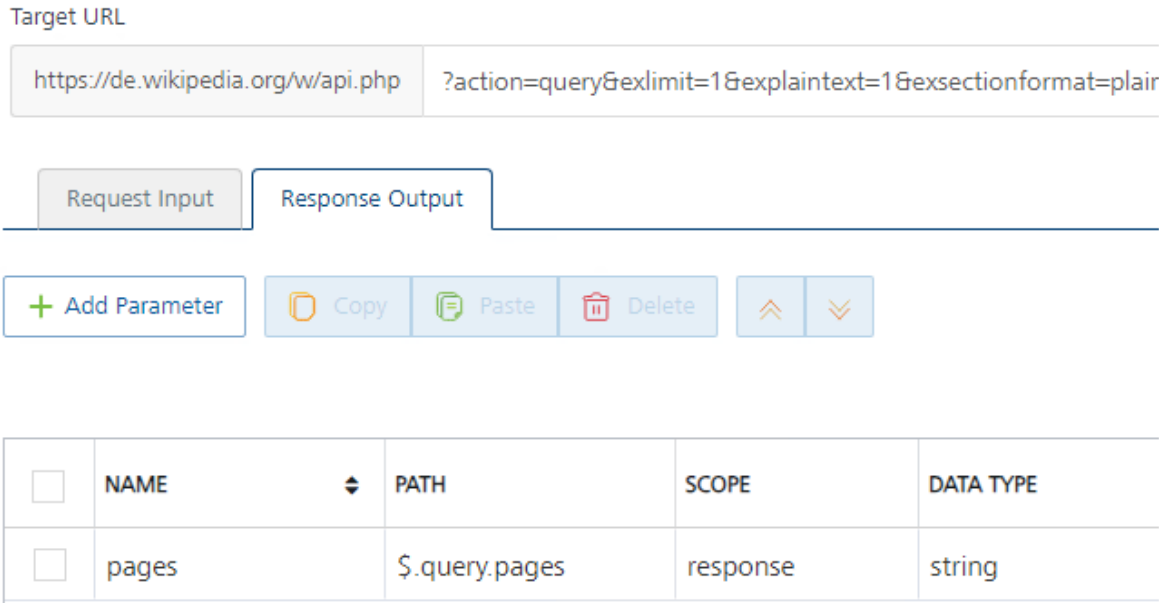
Add a **GET Operation** named **Search** with the following parameters:

URL suffix:
?action=query&exlimit=1&explaintext=1&exsectionformat=plain&prop=extracts&titles=$searchFor&format=json

Request Input for Body: **searchFor**

Response Output NAME: **pages**

Response Output PATH: **$.query.pages**



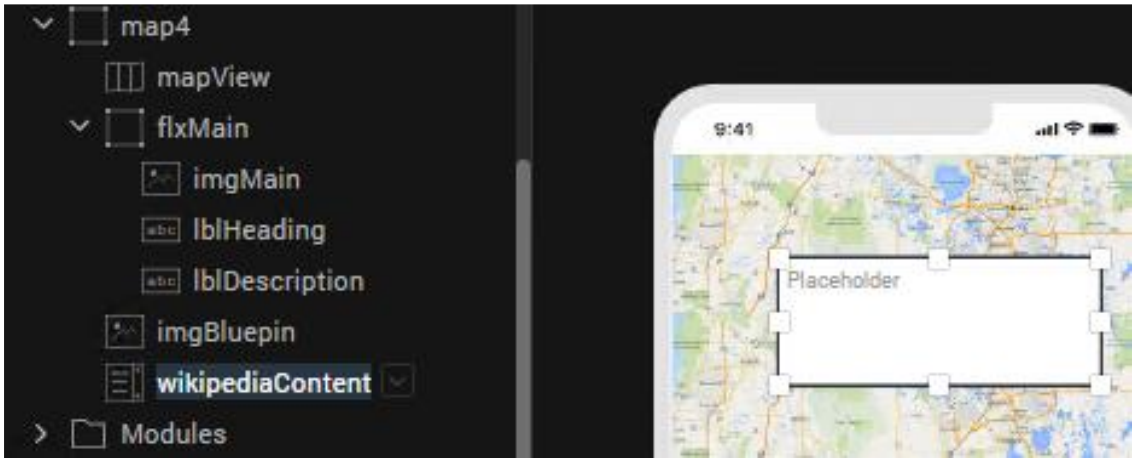
Test your operation with some **TEST VALUE** for **searchFor** (like “HCL”).

If your service works fine, **PUBLISH** it to the Foundry runtime environment.

Switch back to Iris.

From **Form1**, right-click on **map4** and select **Edit Component**.

Drag & drop a **TextArea** component into the **map4** element. Name it **wikipediaContent**.



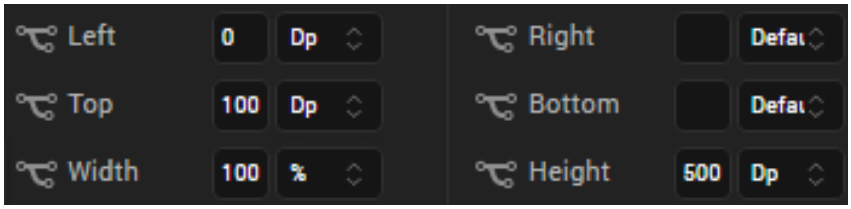
Assign the following positioning properties to that element:

Left: 0

Top: 100dp

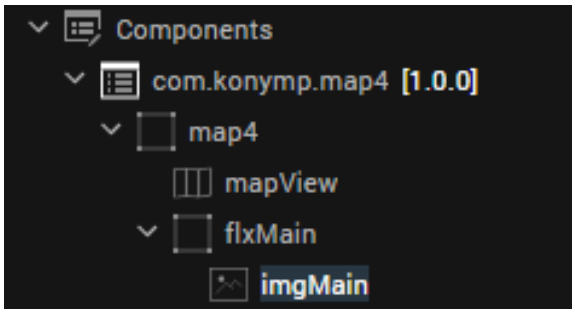
Width: 100%

Height: 500dp



From the **Look** tab, set the **Visibility** of that element to **Off**.

Find the **imgMain** image.

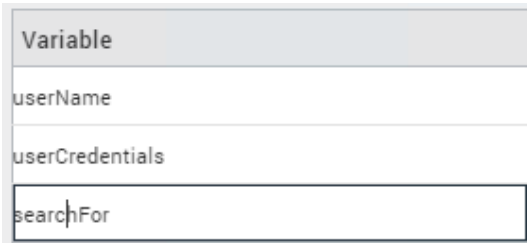


There is an **onTouchStart** method that we extend by an **Add Snippet**. This snippet will consume our Foundry service, retrieve data and then enable the UI element to show that data.

Add the following code here:

```
function INVOKE_SERVICE_f120f7dc841d4adea8ae0141b24cec06_Callback(status, Search) {
  voltmx.application.dismissLoadingScreen();
  if (Search.opstatus === 0) {
    var searchResult = Search.pages;
    if (searchResult.indexOf("extract=") !== -1) {
      searchResult = searchResult.substring(searchResult.indexOf("extract=") + 8);
      searchResult = searchResult.substring(0, searchResult.indexOf("}}"));
    } else {
      searchResult = "Keine Ergebnisse gefunden";
    }
    self.view.wikipediaContent.text = searchResult;
    self.view.wikipediaContent.isVisible = true;
  }
}

var Search_inputparam = {};
Search_inputparam["serviceID"] = "userXWikipediaService$Search";
var Search_httpheaders = {};
Search_inputparam["searchFor"] = searchFor;
Search_inputparam["httpheaders"] = Search_httpheaders;
var Search_httpconfigs = {};
Search_inputparam["httpconfig"] = Search_httpconfigs;
userXWikipediaService$SearchD = mfintegrationsecureinvokerasync(Search_inputparam,
"userXWikipediaService", "Search",
INVOKE_SERVICE_f120f7dc841d4adea8ae0141b24cec06_Callback);
```

Add another global variable **searchFor** to your application.



To set this variable, oben the **map4Controller.js** module again. Add the following to the **pinClicked** method:

```
searchFor = location.lblheading;
```

Test your App. It should now embedd Wikipedia information for any location you select. Just click on the pin and then touch the location's image.

Congratulations! You have extended your mobile app to visualize data from Domino in a completely different UI experience!

References

HCL Volt MX Trial

https://manage.demo-hclvoltmx.com/registration

HCL Volt MX for Domino Developers Self-Paced Tutorial

https://hclsoftwareu.hcltechsw.com/hclsoftwareu-courses/course/volt-mx-for-domino-developers-sp-tutorial

Domino Access Services

https://ds-infolib.hcltechsw.com/ldd/ddwiki.nsf/xpAPIViewer.xsp?lookupName=IBM+Domino+Access+Services+9.0.1#action=openDocument&res_title=Accessing_IBM_Domino_Access_Services_das901&content=apicontent

Domino REST API

https://opensource.hcltechsw.com/Domino-rest-api/index.html

Legal Statements

This edition applies to version 9 of HCL Volt MX Iris.

When you send information to HCL Technologies Ltd., you grant HCL Technologies Ltd. a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Disclaimers

This report is subject to the HCL Terms of Use (https://www.hcl.com/terms-of-use) and the following disclaimers:

The information contained in this report is provided for informational purposes only. While efforts were made to verify the completeness and accuracy of the information contained in this publication, it is provided AS IS without warranty of any kind, express or implied, including but not limited to the implied warranties of merchantability, non-infringement, and fitness for a particular purpose. In addition, this information is based on HCL's current product plans and strategy, which are subject to change by HCL without notice. HCL shall not be responsible for any direct, indirect, incidental, consequential, special or other damages arising out of the use of, or otherwise related to, this report or any other materials. Nothing contained in this publication is intended to, nor shall have the effect of, creating any warranties or representations from HCL or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of HCL software.

References in this report to HCL products, programs, or services do not imply that they will be available in all countries in which HCL operates. Product release dates and/or capabilities referenced in this presentation may change at any time at HCL's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way. The underlying database used to support these reports is refreshed on a weekly basis. Discrepancies found between reports generated using this web tool and other HCL documentation sources may or may not be attributed to different publish and refresh cycles for this tool and other sources. Nothing contained in this report is intended to, nor shall have the effect of, stating.

or implying that any activities undertaken by you will result in any specific sales, revenue growth, savings or other results. You assume sole responsibility for any results you obtain or decisions you make as a result of this report. Notwithstanding the HCL Terms of Use (https://www.hcl.com/terms-of-use), users of this site are permitted to copy and save the reports generated from this tool for such users own internal business purpose. No other use shall be permitted.