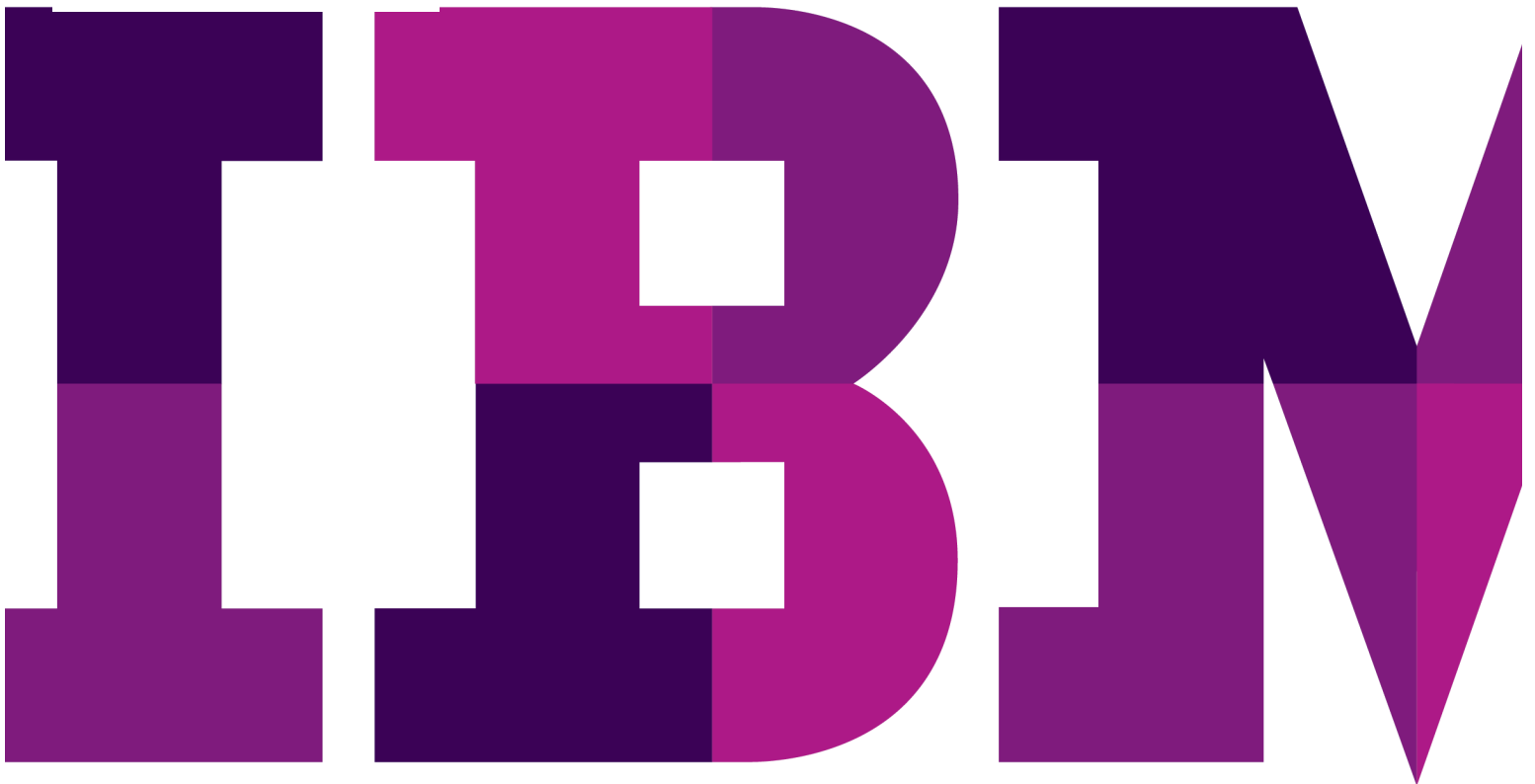


16055 - IBM Mobile Platform development hands on lab

Lab Exercise



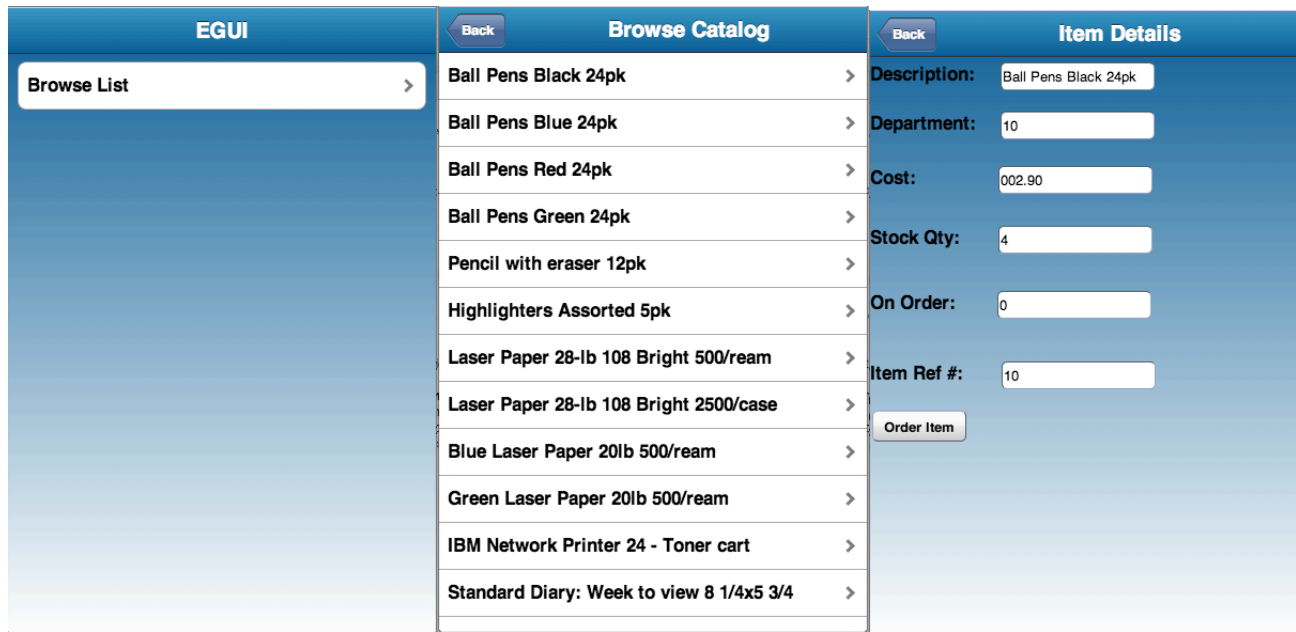
Contents

1	GETTING STARTED WITH IBM WORKLIGHT.....	4
1.1	START THE WORKLIGHT STUDIO	5
1.2	CREATE THE CICS MOBILE PROJECT AND HELLO WORKLIGHT APPLICATION	7
1.2.1	CREATE THE CICS MOBILE WORKLIGHT PROJECT AND EGUIXX APPLICATION.	7
1.2.2	CONNECT TO WORKLIGHT SERVER ON STARTUP	11
1.3	COMMON RESOURCES IN WEB PREVIEW	13
1.4	EXPLORING WORKLIGHT ENVIRONMENTS.....	14
1.5	ENHANCING THE EGUI APPLICATION WITH CONTENT	19
1.5.1	ADDING THE VIEWS	19
1.5.2	ADDING THE STRUCTURE TO THE VIEWS.....	25
1.5.3	ADDING FUNCTIONALITY TO THE VIEWS	29
1.6	CONNECTING TO CICS USING WORKLIGHT ADAPTER	30
1.7	TEST WITH MOBILE BROWSER SIMULATOR.....	39
1.8	SUMMARY	43

1 Getting Started with IBM Worklight

In this lab you will develop a basic mobile app using the IBM Worklight Studio development environment. You will use cross platform techniques such as HTML5, CSS3, JavaScript and the Dojo Mobile framework. The mobile application you will develop and use throughout this will be called EGUI and its purpose is to show you how you can quickly develop a mobile front end to interface and existing 3270 based CICS application running on z/OS systems. The application that you are going to connect to is the CICS Catalog Manager sample application.

Below are some screenshots from the fully developed application in this lab:



In this lab you will build a fully functional version of these three views in the rich page editor.

Upon completion of this exercise you should have gained basic understanding of

- How to create a Worklight project and a Worklight application in Worklight Studio
- How to build and deploy a Worklight application to the test server in the Worklight Studio
- How to preview and test an application from the Worklight Studio
- How to create Worklight Environments for platforms such as Android, iPhone, etc.
- How to use the Rich Page Editor to add UI elements to an application
- How to create and implement a Worklight Adapter (HTTP)
- How to run an application as a Web app
- How to test an application using the Mobile Browser Simulator

You should possess basic knowledge of HTML, CSS and JavaScript. Familiarity with the Eclipse platform is a plus, but not required. Throughout the lab, it is recommended to save your work routinely.

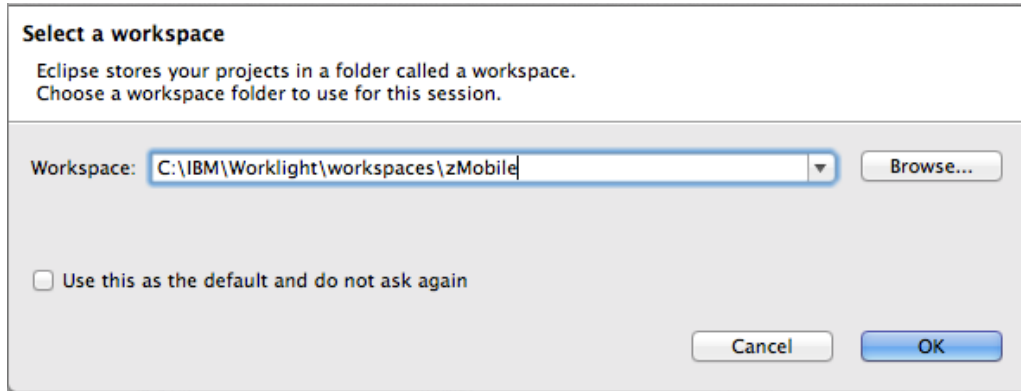
Here are some important values to be aware of for the lab.

Field	Entry
<LAB_HOME>	Root directory will be provided by instructor
Worklight Studio	<LAB_HOME>/IBM/Worklight/WorklightStudio/eclipse.exe
Android SDK location	<LAB_HOME>/IBM/Worklight/utils/android-sdk
Lab snippets	<LAB_HOME>/IBM/Worklight/imports
workspace.dir	<LAB_HOME>/IBM/Worklight/workspaces/zMobile
cics.server.address	Provided by instructor
cics.server.port	Provided by instructor
Worklight AppName	Provided by instructor (EGUIxx)

1.1 Start the Worklight Studio

This lab assumes that you have obtained and started a corresponding VMWare image or running the software package on a native desktop. In the image you will launch Eclipse with the Worklight Studio tooling and then create a project for the EGUI app.

- __1. Starting the Worklight Studio can be done either via a shortcut or directly opening the eclipse application as shown below.
 - __a. On the Desktop, double-click the **Worklight Studio** icon (shown below).
 - __b. Open the eclipse.exe application located under the <LAB_HOME>/IBM/Worklight/WorklightStudio/ directory.
- __2. On the Workspace Launcher dialog accept the default workspace path **{workspace.dir}** and click **OK**. Note the workspace directory provided by the instructor.



- __3. An Eclipse Welcome Screen will open; dismiss it by **closing** the *Welcome* tab. You may also be prompted to **update** Android SDK or platform tools. You can safely ignore these prompts by canceling any requested actions.

The provided workspace has already been preconfigured with Android tools and various other preference customizations such as external browser (Google Chrome), etc. If you do not use the provided workspace directory there may be some differences in user experience than what the instruction will indicate.

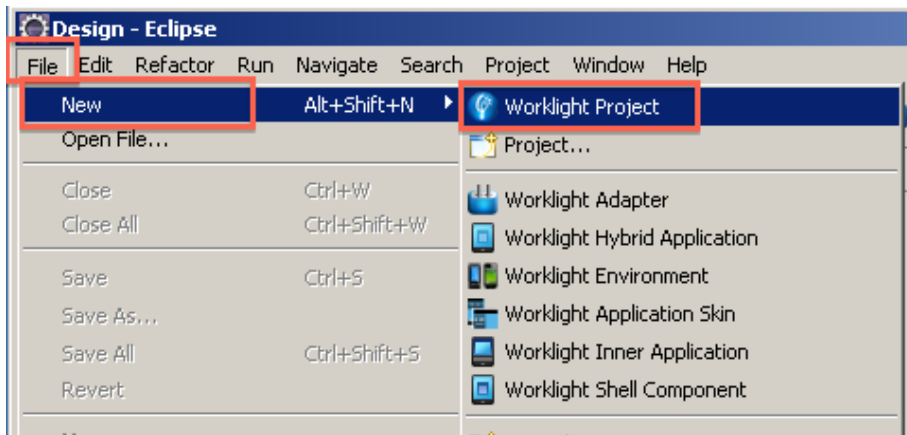
- __4.

1.2 Create the CICS Mobile project and Hello Worklight application

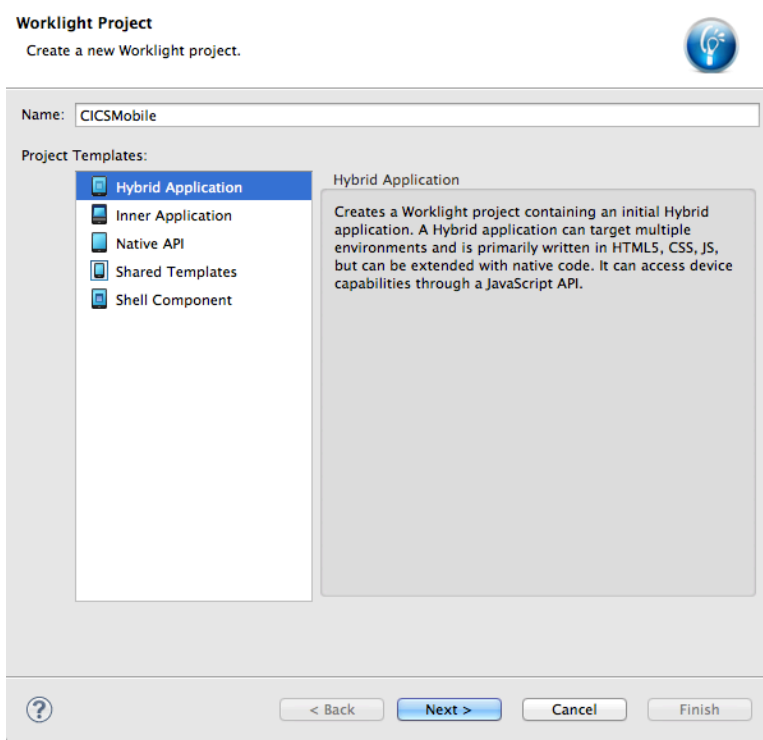
The mobile application that you are building will be connecting to a CICS sample application shipped with the product. The application name CICS Catalog Manager sample application transaction is EUGI, so we will call the mobile application EGUI.

1.2.1 Create the CICS Mobile Worklight Project and EGUIXX application.

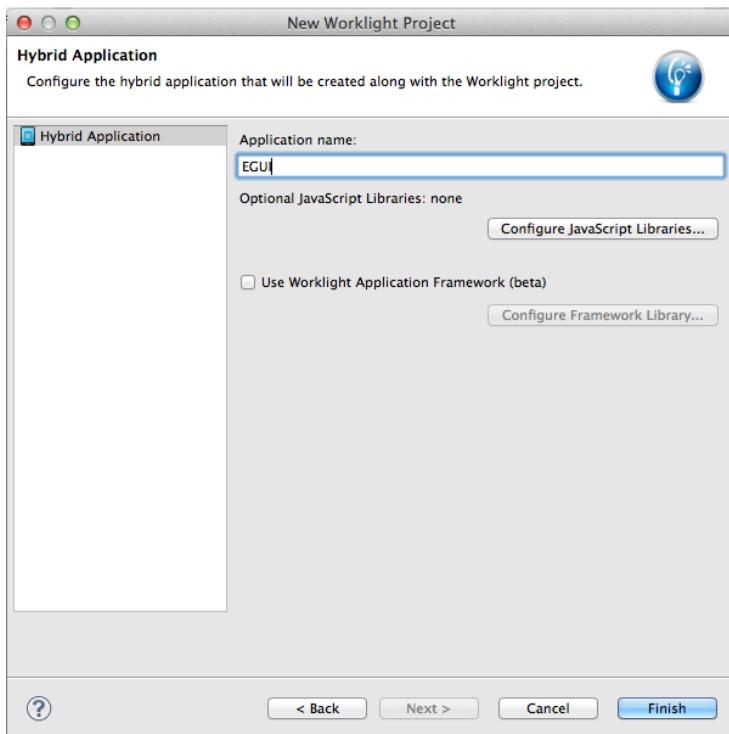
1. In the Eclipse menu, select **File > New > Worklight Project**.



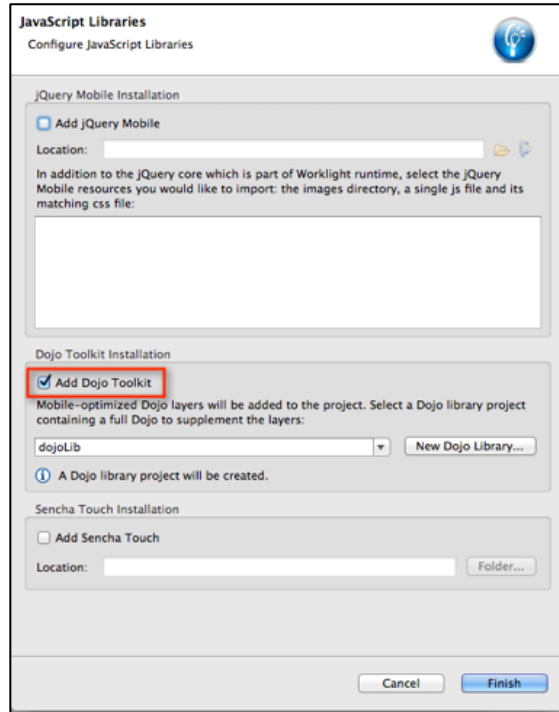
2. Enter **CICSMobile** as the project name, keep the default Project Templates selection for **Hybrid Application** then click **Next**.



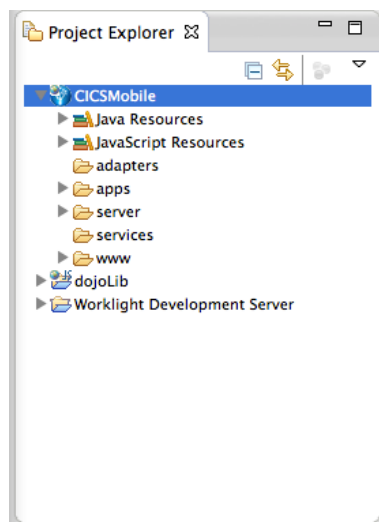
- __3. The Hybrid Application panel will surface noting that you have not specified an Application name. Enter **EGUIxx** (where **xx** is your assigned team number, for example **EGUI01**, **EGUI02**, etc) for the Application name then click on the **Configure JavaScript Libraries...** button.



- __4. In the JavaScript Libraries window, select the **Add Dojo toolkit** check box then click **Finish**.



- __5. Now click the **Finish** button in the Hybrid Application window. The application template will be populated and the **index.html** file will open by default. We can **leave it at its defaults** for now, while we investigate the parts of a Worklight project and application.
- __6. In the *Project Explorer* pane, expand the **CICSMobile** project. Review the folder structure that has been created.



Java Resources

Worklight Server, JRE Sytem and WAS libraries

JavaScript Resources

Contains the project's JavaScript classes content

adapters:

Contains the project's adapters (used for backend connectivity)

apps:

Contains the project's applications

bin: (seen after building content)

Location for build artifacts (wlappp files) that are deployed to a Worklight server

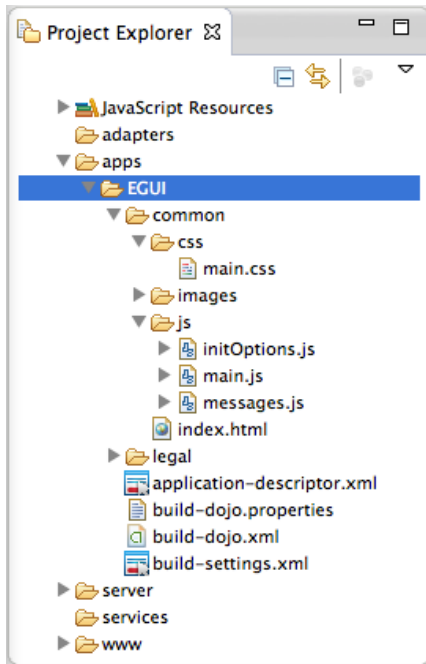
www

Contains the Dojo Toolkit JavaScript source

server

Contains configuration files and extension locations for the embedded Worklight test server

- __7. In the *Project Explorer*, expand the **apps** folder then the **apps > EGUI** folder and the apps > EGUI > common folder that were created by the new application wizard.



common: the default 'environment' that gets created for an application.

css: **main.css** – the main application CSS file

images: Default images for the common environment.

js: **main.js** – the main JavaScript file for the app,
messages.js – JSON object holding all app messages,
auth.js – authentication mechanism.

index.html: The main application html file. Application can have multiple html files

legal: All legal related documents.

application-descriptor.xml: Application's meta data (environments, security config, etc...)

build-dojo*: Artifacts related to custom dojo profile builds (advanced)

- __8. Open the **application-descriptor.xml** file, if not already opened. Switch to the **Source** tab. The following section specifies the application name, description and author's name to be displayed in the Worklight Console.

```

application-descriptor.xml
<?xml version="1.0" encoding="UTF-8"?>
    <!-- Licensed Materials - Property of IBM
    5725-I43 (C) Copyright IBM Corp. 2006, 2013. All Rights Reserved.
    US Government Users Restricted Rights - Use, duplication or
    disclosure restricted by GSA ADP Schedule Contract with IBM Corp. -->
    <!-- Attribute "id" must be identical to application folder name -->
    <application id="EGUI" platformVersion="6.1.0.00.20131126-0630"
    xmlns="http://www.worklight.com/application-descriptor"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <displayName>EGUI</displayName>
        <description>EGUI</description>
        <author>
            <name>application's author</name>
            <email>application author's e-mail</email>
            <copyright>Copyright My Company</copyright>
            <homepage>http://mycompany.com</homepage>
        </author>
        <mainFile>index.html</mainFile>
        <thumbnailImage>common/images/thumbnail.png</thumbnailImage>
        <features>
        </features>
    </application>
  
```

- __9. Environment specific information, such as Android and iPhone, will be inserted automatically as new 'environments' are added to the project. You can observe this change as environments are added during later portions of the lab.

```

34     <android version="1.0">
35         Uncomment and update push sender ID in order to use push notifications for android
36         <pushSender key="keyTest" senderId="senderIdTest" />
37     </android>
38     -->

```

__10.

1.2.2 Connect to Worklight Server on startup

We will now add initialization logic for the application to connect to, and handle a connection failure exception when trying to communicate with the Worklight server

__1. Expand the **common** then **js** folder, and **double-click** to open the **initOptions.js** file.

```

var wlInitOptions = {
    // # Should application automatically attempt to connect to Worklight Server on application start up
    // # The default value is true, we are overriding it to false here.
    connectOnStartup : false,

    // # The callback function to invoke in case application fails to connect to Worklight Server
    //onConnectionFailure: function (){};
}

```

This file contains the parameters needed to connect to the Worklight server. These will be invoked automatically once the Worklight framework initialization completes on the client side. Here we will change the parameters to

- __1. Enable the mobile application to connect with the Worklight server by setting property *connectionOnStartup* : **true**,
- __2. Uncomment the *onConnectionFailure* property and add the following implementation to the call back function (){...}. This will instruct the application to initialize and run in offline mode until connectivity is available.

```

WL.Logger.debug('Unable to connect, running offline!');

wlCommonInit();

```

If desired, you can **Copy and Paste** code from above or **Snippet#1.txt** from the accompanying lab snippets.

Below is a screenshot of what the result will look like after your changes.

```

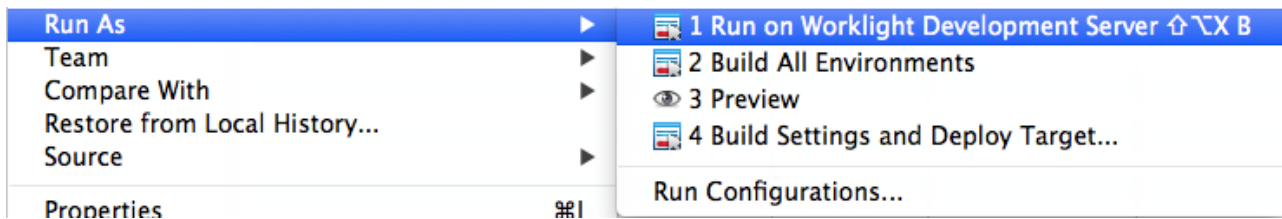
var wlInitOptions = {
    // # Should application automatically attempt to connect to Worklight Server on application start up
    // # The default value is true, we are overriding it to false here.
    connectOnStartup : true,

    // # The callback function to invoke in case application fails to connect to Worklight Server
    onConnectionFailure: function (){
        WL.Logger.debug('Unable to connect, running offline!');
        wlCommonInit();
    },

    // # Worklight server connection timeout
    //timeout: 30000,

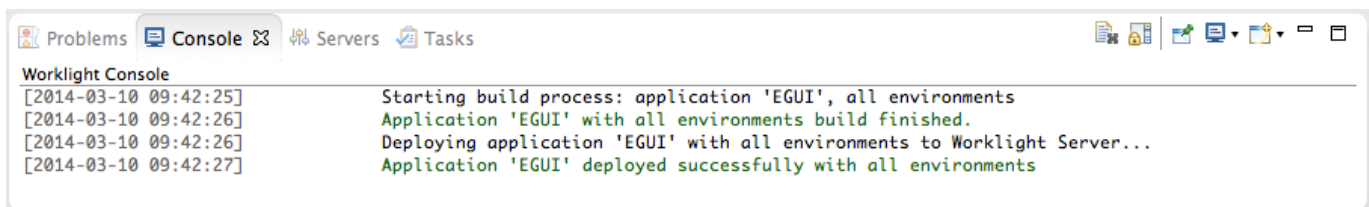
```

4. Save the initOptions.js file
5. Perform a build and deploy of the work that you have completed so far by following these steps.
 - a. Right-click the **EGUI** application and select **Run As > Run on Worklight Development Server**.



This step will build and publish your application to an embedded test server within eclipse, where we can preview and test as part of the development life cycle. This step may take some time to complete, as the embedded Worklight Server will need to be started if not already running. Depending on the load that your machine is under, response time may vary.

- b. Verify that the build process and that the deployment to Worklight server was successful by examining the **Worklight Console** log output in the Console view.



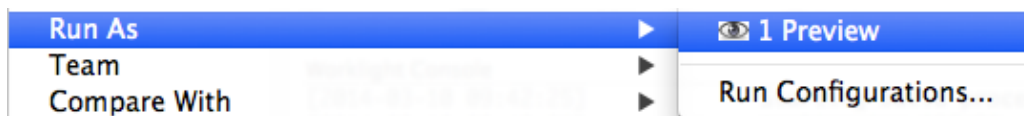
If the console is not displayed, open it with **Window > Show view > Other > Console**.

Use Window > Use the console icon () to switch between the various consoles if necessary, until you find the **Worklight Console**.

1.3 Common resources in web preview

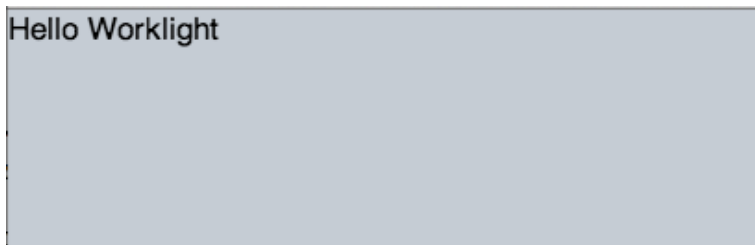
For this lab, we have configured eclipse to use an external web browser in the workspace preference, which opens the browser window outside of eclipse. Using an external browser gives freedom to choose your rendering browser, access to advanced debugging features like Web Inspector in Chrome or Firebug with Firefox, and access to internet settings, cache and history that are not available when running with the internal browser setting in eclipse (see Window > Preferences > General > Web Browser for this setting).

1. Right-click on either the **common** folder (to run the common resource web app) or the **index.html** file within the common folder and select **Run as > Preview** (use the first preview option in the menu)



2. A new external browser window will open rendering our EGUI application as a mobile web application.

As this application has no device characteristics yet, a simple, un-styled mobile browser view is rendered, with no device configuration options.



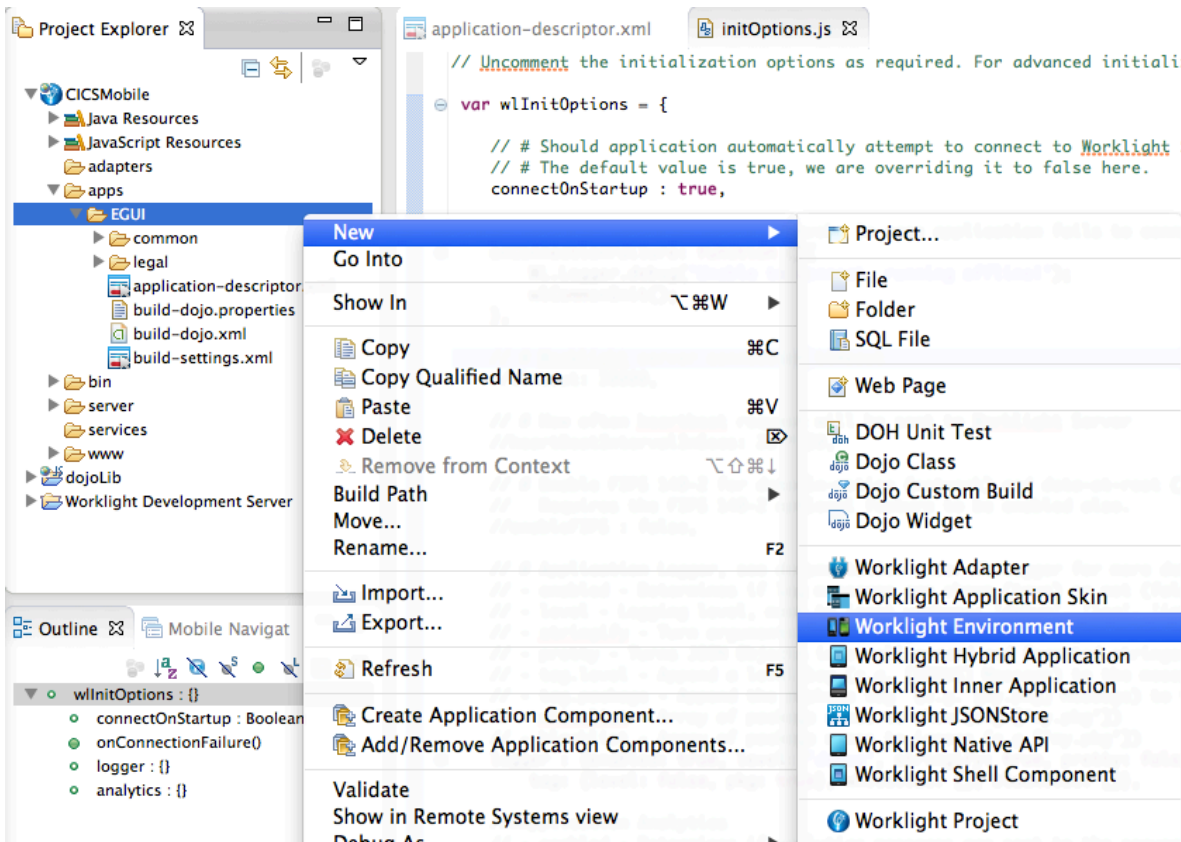
Congratulations! You are previewing your first hello world-like application developed using Worklight.

1.4 Exploring Worklight Environments

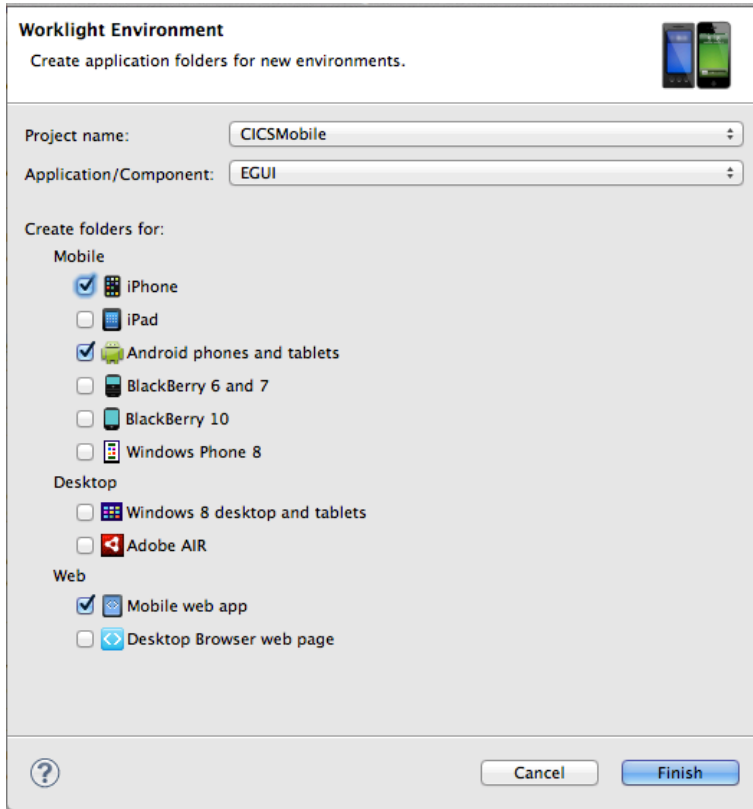
An environment is a mobile, desktop, or web platform capable of displaying web-based applications, such as the Apple iPhone, Android phones, Windows Phone 8, and BlackBerry among others. In this section you will create environments to provide support for iPhone, Android and Mobile Web.

__1. Creating the Worklight Environments for the EGUI application

- __a. In the *Project Explorer* select the **EGUI** application (in the /apps/EGUI folder). In the right click menu select **New > Worklight Environment**.



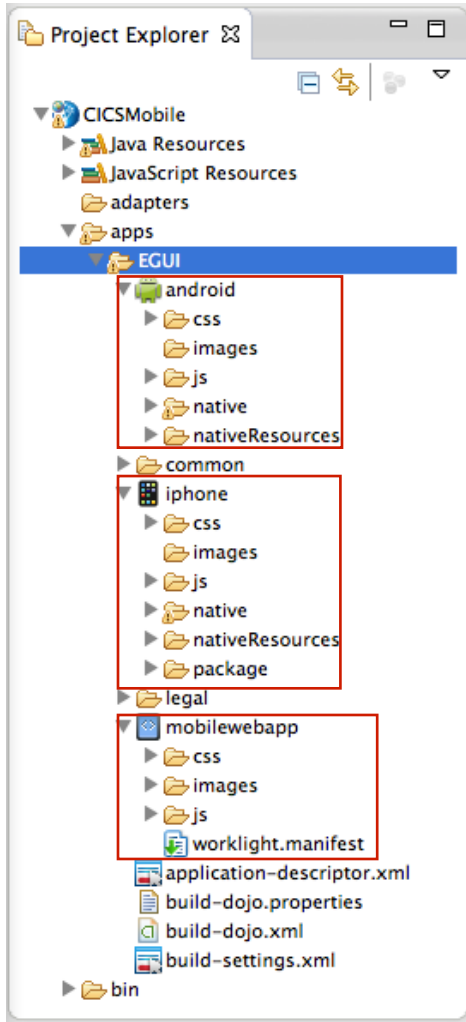
- __b. In the *Worklight Environment* dialog, select **iPhone, Android phones and tablets**, and **Mobile web app** (as shown below) then click **Finish**.



- ___c. Observe the *Console* window, notice that the messages about the environments that you have just chosen.



- ___d. In the **Project Explorer**, observe that there are now additional folders created under the **EGUI** application folder.

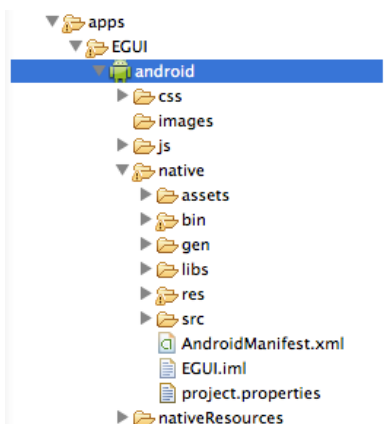


The new environment's resources will have the following relationship with the common resources:

- **images** - override the common images in case both share the same name
- **css** – extend and/or override the common CSS files
- **js** - extends the common application instance JS object (The environment class extends the common app class)
- **HTML** - override the common HTML code in case both share the same name
- **native:** contains environment specific generated application code

__2. Quick review of the native folder for specific environments

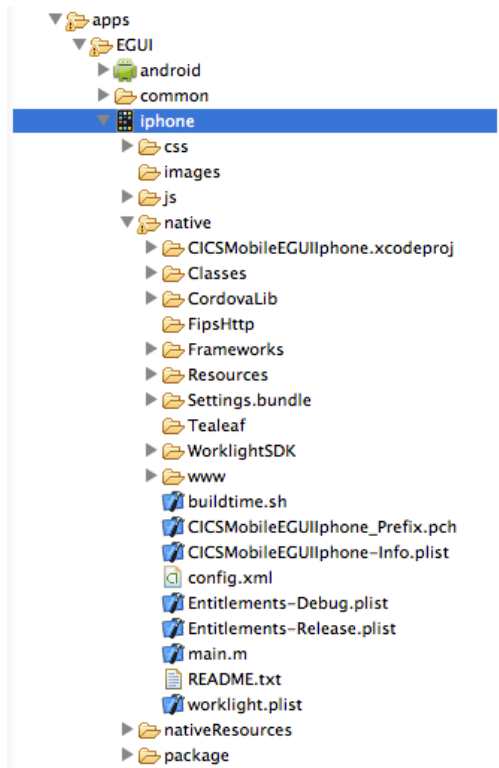
__a. android native folder



The **native** folder under android contains automatically generated android application code that is imported into the eclipse workspace as an Android Project during the Build and Deploy step (later).

It is not recommended to edit files under the assets folder, as each time the application is built they are regenerated.

__b. iPhone folders



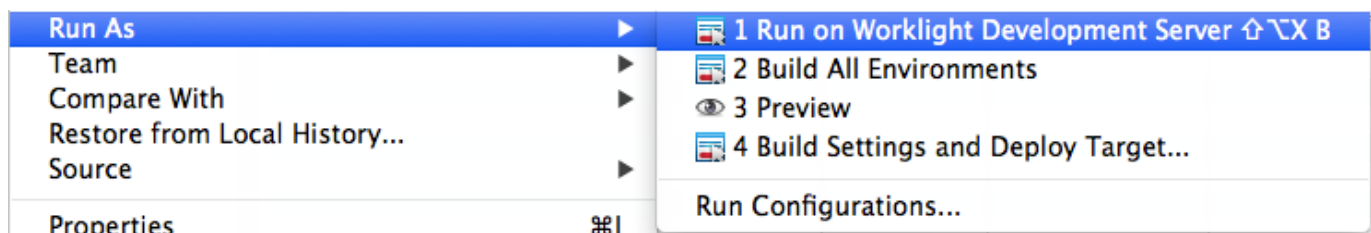
The **native** folder under iphone contains automatically generated iphone app code

The **package** folder under iphone contains a packaged (zipped) application

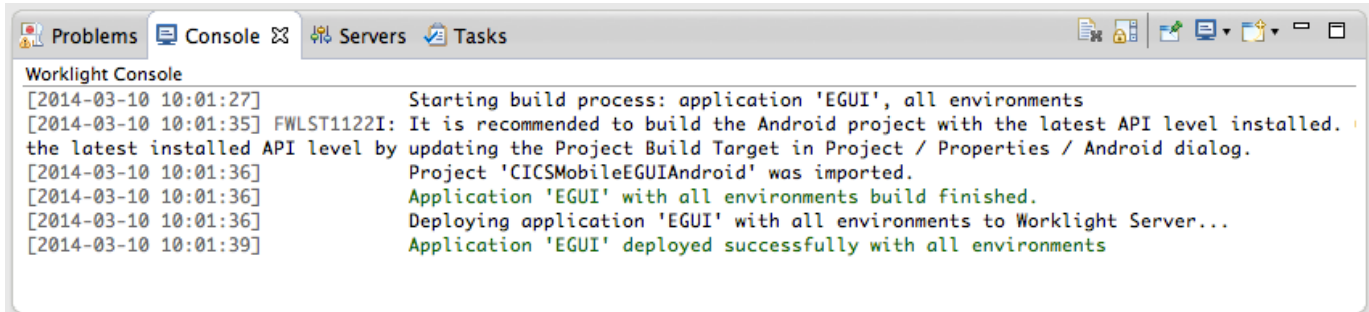
It is not recommended to edit files under native/www folder, as each time the application is built they are regenerated

__3. Build and deploy all environments

__a. Right-click the **EGUI** application and select **Run As > Run on Worklight Development Server**.

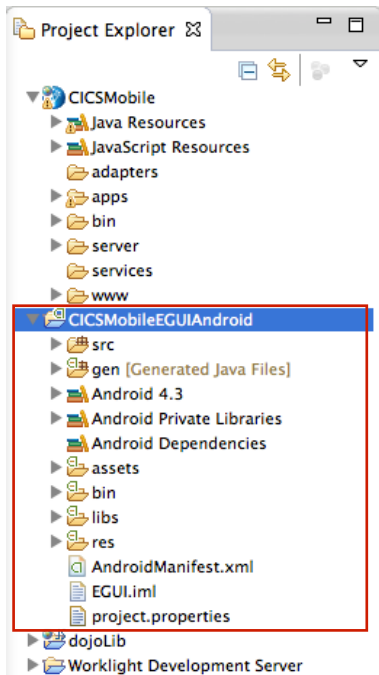


__b. Check the Console log to make sure that the build and deploy is successful.



- c. Observe that the Native Android project has been created and imported into Eclipse workspace.

This is an example of IBM Worklight's tight integration with the device SDKs. Because the Android SDK is Eclipse-enabled, Worklight is able to immediately generate and build a native Android Project in the Eclipse workspace (same for BlackBerry). For other platforms, Worklight will launch the respective non-Eclipse tooling and provide a project in that tool's format for completion (eg. Xcode).



We will add some more interesting content in the next section before previewing the environment-specific variations of the application.

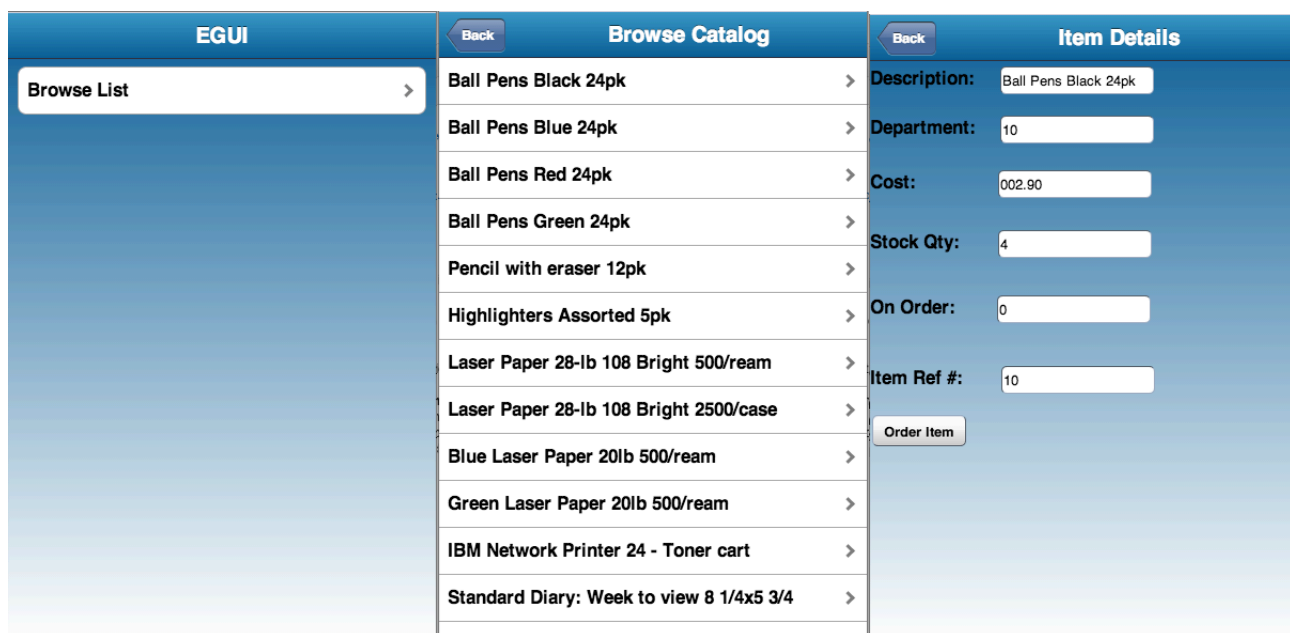
1.5 Enhancing the EGUI application with content

At this point, the EGUI application is now capable of being built, compiled and deployed to run on Android and iPhone devices as well as mobile and web browser. However, there isn't much to see or do as it still a Hello World application. This section of the lab guide will provide instruction for adding visual and functional content to the application. For purposes of available time and interest, some portions of the content will be provided as lab snippets in order to demonstrate the capabilities of the tooling without going through a tedious coding exercise. Feel free to augment and change as seen fit after completing the provided steps.

1.5.1 Adding the views

In this part of the Lab, we are going to continue developing our EGUI application using the Rich Page Editor visual editor commonly referred to as 'what you see is what you get' or WYSIWYG.

Review the content you will build - three mobile application views: The completed Main, Browse Catalog, and Item Details views of the EGUI mobile application that will be built in this lab are shown below:



The Main view consists of a `dojox.mobile.ScrollableView`, with a Header, and button (Browse List).

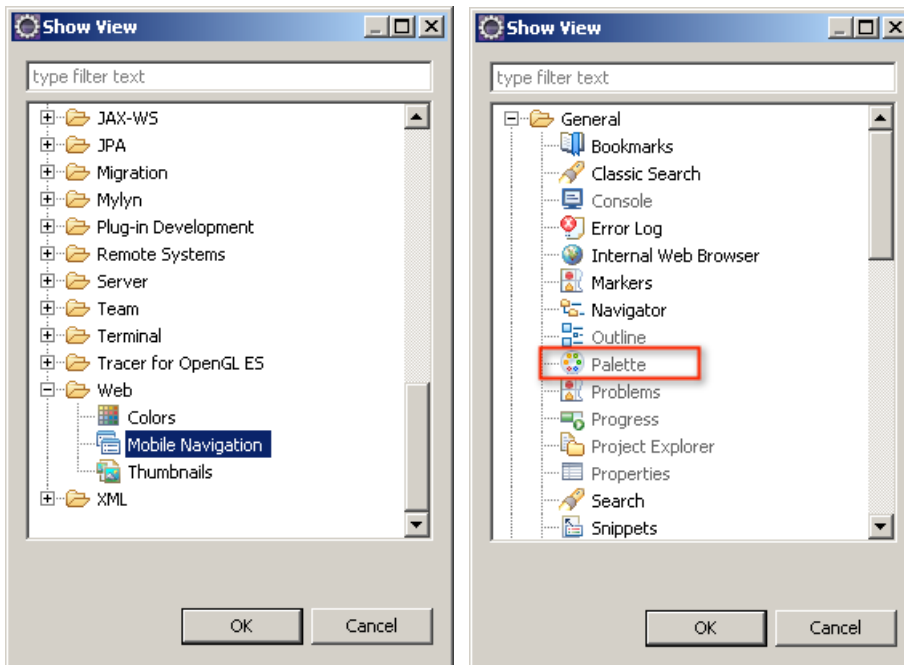
The Browse List view is also built on a `dojox.mobile.ScrollableView`, and contains a Header, a navigation button to return to the Main view, and a scrollable listing of the items in the catalog. When one of the items in the list is selected, you are directed to the Item Details View.

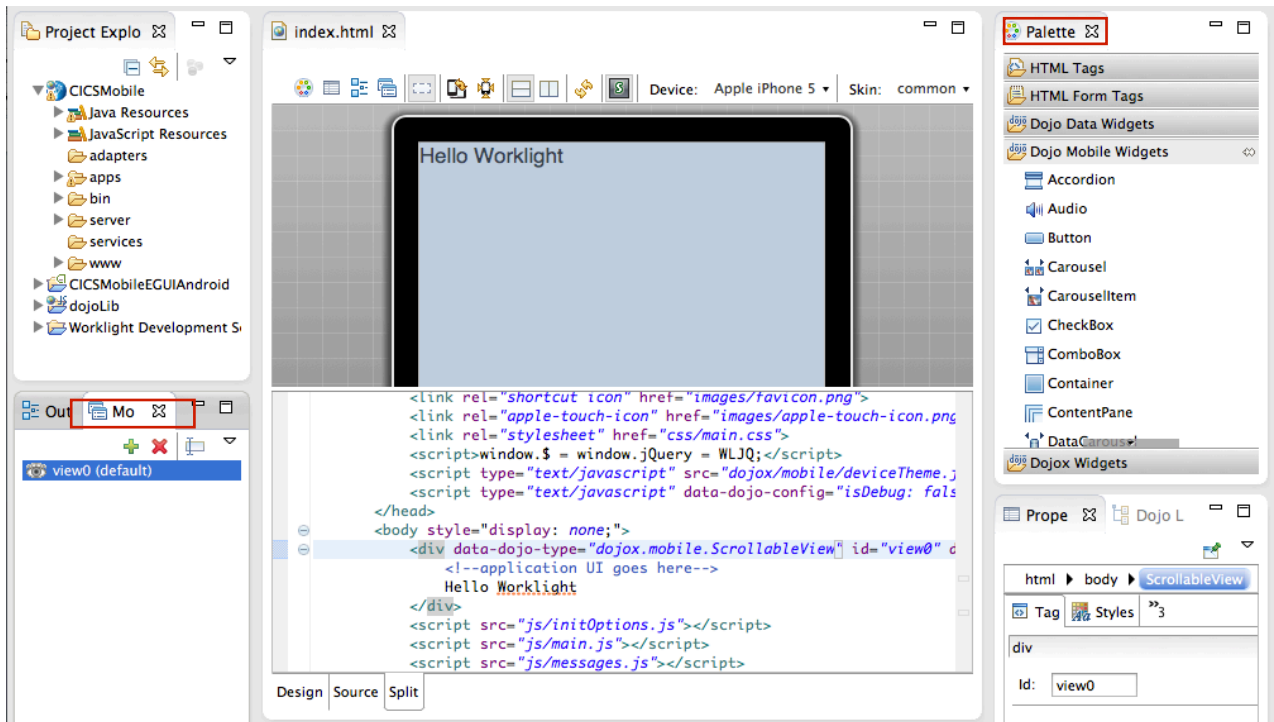
When an item is selected via the Browse List View or the Inquire Items View, you are directed to the Item Details View. This view is also a `dojox.mobile.ScrollableView`, with the item details listed. This view also has six `dojox.mobile.TextBox(s)` and an 'Order Items' Button. When the items quantity is selected and the 'Order Items' button is tapped, that item and its quantity is ordered.

- __1. Prepare Eclipse for Rich Page Editor development by opening the appropriate eclipse Views.

Ensure that you have both the **Palette** and **Mobile Navigation** tabs visible for use with the Rich Page Editor. We will use Mobile Navigation to manipulate and navigate between the ScrollableViews in index.html, and the Palette will allow us to drag visual elements onto the page.

Make sure that you are in the Design perspective, if not already. If these views are not present, in the Eclipse menu navigate to **Window** → **Show View** → **Other...** to display the list of available views.



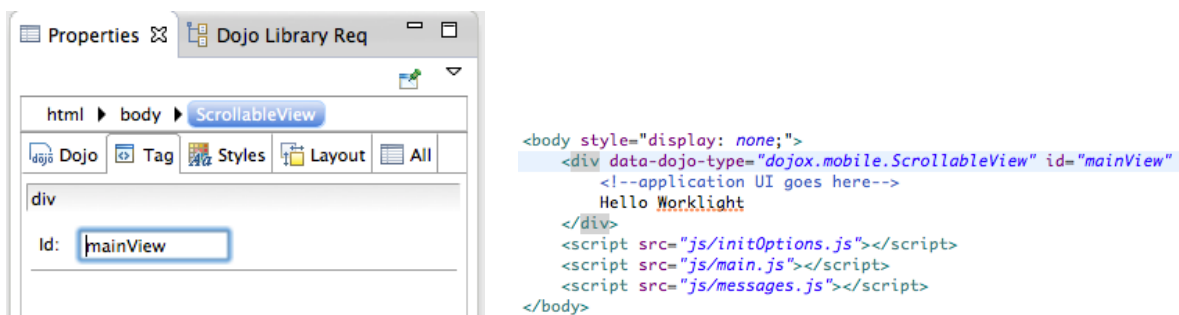


(You can place the tabs in whichever way seems most efficient to you)

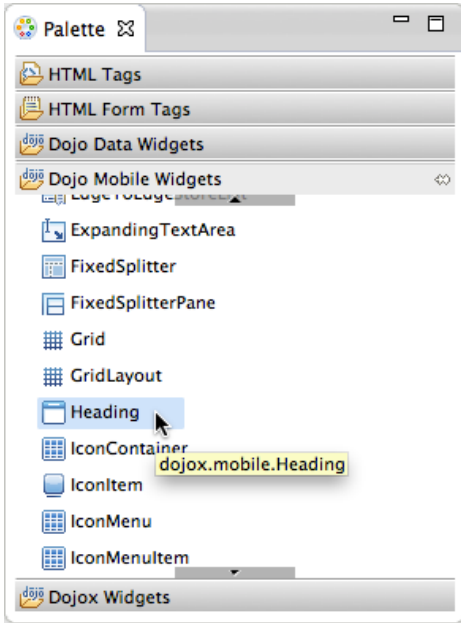
“Pages” in a mobile application are built on views. For this application we are using `dojox.mobile.ScrollableViews`, which are dojo view widgets that manage a view pane. Ensure that the `index.html` file is the active file in the editor pane, and that the design or split editor tab is visible.

NOTE: Sometime it may be necessary to close and re-open the `index.html` file if the Design pane does not render the content as expected.

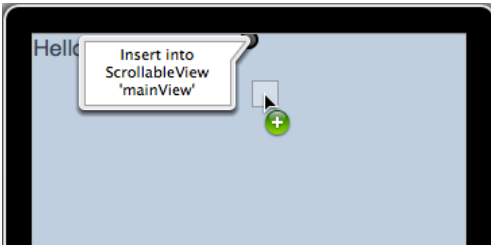
- ___2. Open the file `index.html` into the editor if it's not already opened. You will be updating the default `ScrollableView` in the `index.html` page. This view was added by default by the template tools when you select the Dojo toolkit in the beginning of the first section
- ___3. Update the **Id** of the view to **mainView** by changing the html source directly or using the Properties view.



- ___4. Add a Heading to the **mainView** by inserting a `dojox.mobile.Heading` widget into the `ScrollableView` `mainView` from the Dojo Mobile Widgets Palette.

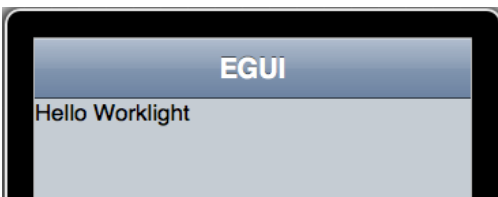


NOTE: Make sure to pay attention to where the widget is placed, based on the popup message shown while hovering over the view. In this case the Heading widget is being inserted into the view name 'mainView'.



__5. Update the heading label in the html source to EGUI.

```
data-dojo-props="label:'EGUI', fixed:'top'"
```



__6. Remove the Hello Worklight default text from the view content.

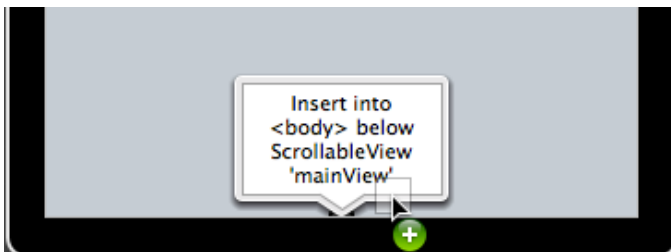
```

<body style="display: none;">
  <div data-dojo-type="dojox.mobile.ScrollableView" id="mainView">
    <!--application UI goes here-->
    Hello Worklight
  </div>
  <script src="js/initOptions.js"></script>
  <script src="js/main.js"></script>
  <script src="js/messages.js"></script>
</body>

```

- __7. Next we'll add a second dojox.mobile.ScrollableView to our application by dragging another ScrollableView widget from the Dojo Mobile Widgets section of the Pallet into the <body> of our index.html page, **below** the mainView ScrollableView as follows.

NOTE: To do this, we need to be scroll down to be bottom of the device screen in the design portion of the split panes.



- __8. After dragging and dropping, the 'Create a new Dojo Mobile Scrollable View' dialog will surface.
- __a. Enter `browseList` as the **Id**.
 - __b. Select the **Include heading** checkbox to include a heading.
 - __c. Enter `Browse Catalog` as the **Heading label**.
 - __d. Enter `Back` as the **Back button label** to add a Back button label.
 - __e. Select `mainView` from the **Back button target** drop-down box.
 - __f. Click **Finish** to generate the ScrollableView code for our Browse List view.

Create a new Dojo Mobile Scrollable View
Create a new Dojo Mobile Scrollable View

Id:

Set as default view Scroll direction:

Include heading

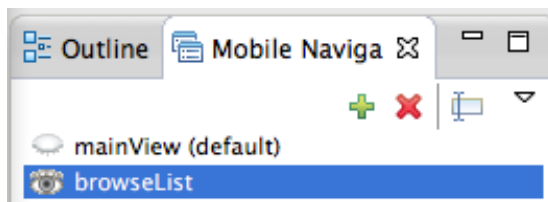
Heading details

Label:

Back button label:

Back button target:

__9. In the Mobile Navigation view, double click on the **browseList** to select the view.



__10. Lastly we'll add a third dojox.mobile.ScrollableView to our application by dragging another ScrollableView widget from the Dojo Mobile Widgets section of the Pallet into the <body> of our index.html page, **below** the BrowseList ScrollableView. To do this, make sure that the browseList view is selected in the Mobile Navigation then add the widget towards the bottom of the device screen.



After dragging and dropping, the 'Create a new Dojo Mobile Scrollable View' dialog will surface.

- __a. Enter `itemDetails` as the **Id**.
- __b. Select the **Include heading** checkbox to include a heading.
- __c. Enter `Item Details` as the **Heading label**.
- __d. Enter `Back` as the **Back button label** to add a Back button label.
- __e. Select `browseList` as the **Back button target**.

- __f. Click **Finish** to generate the ScrollableView code for our Item Details view.

- __11. Open the Mobile Navigation tab and notice how you can alternate the contents of the design tab in the Rich Page Editor by selecting which is the active Mobile View – the view with the open eye icon is currently displayed in the design tab. Double click the closed eyelid to make that Mobile View become the visible view in the editor.

If you double click to select the **mainView** in the Mobile Navigation tab, it is rendered in the design tab. This goes for any of the three views we created.

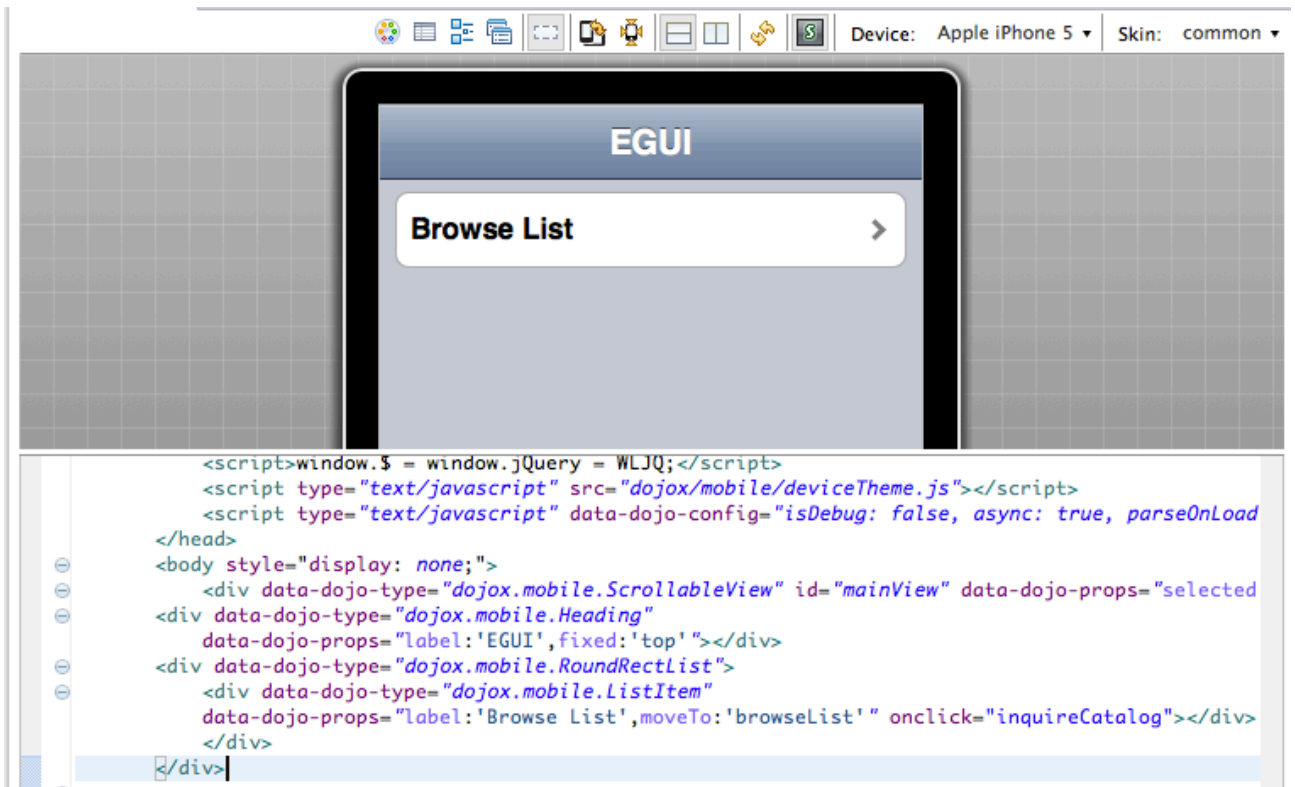
Congratulations, we have now designed our application's three views. We now want to create the content of those views.

1.5.2 Adding the structure to the views

- __1. On the mainView view, we want to add a `dojox.mobile.RoundRectList`, which will house a `dojox.Mobile.ListItem`, (Browse List). When this list item is tapped, we are taken to the BrowseList view.
- __a. From the Mobile Navigation Tab above, we want to make sure that the mainView (default) is selected.
- __b. Drag and drop a `dojox.mobile.RoundRectList` from the Dojo Mobile Widgets section of the Pallet into the 'mainView ScrollableView **below** Heading (as seen below).



You now see that a `dojox.mobile.RoundRectList` has been added with a `dojox.mobile.ListItem` already created. We now want to customize the `dojox.mobile.ListItem` to correspond to the Browse List view.



- __2. Add properties to the `ListItem` widget to make it interactive by either updating the source code directly or use the properties view.
 - __a. Change the label text value to `Browse List`.
 - __b. Add an `moveTo: browseList` property to the `dojox.mobile.ListItem` element, which will take us to the next view.
 - __c. Add an on-click property to the `dojox.mobile.ListItem` element, which will display the list of items to browse.

This is the code after the change:

```
data-dojo-props="label:'Browse List',moveTo:'browseList'"
  onclick="inquireCatalog"></div>
```

You have now designed the mainView view, added a list item that is now linked to the browseList view. We will now design the browseList view.

- ___3. On the browseList view, we want to add a **dojo.mobile.EdgeToEdgeList**, which will house all of the items in the catalog that we want to list. When one of these list items are tapped, we are taken to the corresponding item's **itemDetails** view
 - ___a. From the Mobile Navigation Tab, we want to make sure that the **BrowseList view** is selected.
 - ___b. Drag and drop a **dojo.mobile.EdgeToEdgeList** from the Dojo Mobile Widgets section of the Pallet into the 'BrowseList' ScrollableView **below** Heading (as seen below).



- ___c. We will now make a slight change to the **dojo.mobile.EdgeToEdgeList** in the source view. **Delete** the **dojo.mobile.ListItem** completely, from `</div>` to `<div>` as shown in the red rectangle below.
- ___d. Also **add** element `id="catalogList"` to the **dojo.mobile.EdgeToEdgeList** element. The source code should look like this after the change:

```
<div data-dojo-type="dojo.mobile.EdgeToEdgeList" id="catalogList">
```

Both of these changes are reflected below.

The first image is the before:

```
<div data-dojo-type="dojo.mobile.ScrollableView" id="browseList"
  data-dojo-props="selected:false,scrollDir:'v'">
  <div data-dojo-type="dojo.mobile.Heading"
    data-dojo-props="label:'Browse Catalog',back:'Back',moveTo:'mainView'"
  <div data-dojo-type="dojo.mobile.EdgeToEdgeList">
    <div data-dojo-type="dojo.mobile.ListItem"
      data-dojo-props="label:'Item'"></div>
  </div>
```

This image is the after:

```

<div data-dojo-type="dojox.mobile.ScrollableView" id="browseList"
  data-dojo-props="selected:false,scrollDir:'v' ">
  <div data-dojo-type="dojox.mobile.Heading"
    data-dojo-props="label:'Browse Catalog',back:'Back',moveTo:'mainView',fixed:'top' "></div>
  <div data-dojo-type="dojox.mobile.EdgeToEdgeList" id="catalogList">
  </div>
</div>

```

- __e. Make sure to **save your changes** to the index.html.

This change allows the dojox.mobile.EdgeToEdgeList to be populated with a number of list items dynamically, which we will get to in later steps.

- __4. You have now designed the BrowseList view, added a dojox.mobile.EdgeToEdgeList which will be linked to the itemDetails view. We will now design the itemDetails view.

The ItemDetails view will consist of six textboxes and a button (Order Item). The six textboxes have some description text, and are where the data for each catalog item will go once it is retrieved from the VSAM dataset on our back-end enterprise system

- __a. From the Mobile Navigation Tab, make sure that the **ItemDetails view** is selected.
- __b. Copy the code from the **Snippet#2.txt** file found in the lab snippets folder. You will want to make sure that the code snippet (Snippet#2.txt) is copied in its entirety, and pasted directly after the heading for the ItemDetails view as shown below. The red rectangle surrounds the heading. Note that the open and closing <div> for the view it self will surround the content being pasted.

```

<div data-dojo-type="dojox.mobile.ScrollableView" id="itemDetails"
  data-dojo-props="selected:false,scrollDir:'v' ">
  <div data-dojo-type="dojox.mobile.Heading"
    data-dojo-props="label:'Item Details',back:'Back',moveTo:'browseList',fixed:'top' "></div>
  <label for="desc" style="font-weight: bold"> Description:</label> <input data-dojo-type="dojox.mobile.TextBox"
  <label for="dept" style="font-weight: bold"> Department:</label> <input data-dojo-type="dojox.mobile.TextBox"
  <label for="cost" style="font-weight: bold"> Cost:</label> <input data-dojo-type="dojox.mobile.TextBox" class=
  <label for="stock" style="font-weight: bold"> Stock Qty:</label> <input data-dojo-type="dojox.mobile.TextBox"
  <label for="order" style="font-weight: bold"> On Order: </label> <input data-dojo-type="dojox.mobile.TextBox"
  <label for="itemRef" style="font-weight: bold"> Item Ref. #:</label> <input data-dojo-type="dojox.mobile.TextBo
  <br>
  <button data-dojo-type="dojox.mobile.Button" id="OrderItemButton" style="font-weight: bold" onclick="placeOrder
</div>

```

Make sure to **save the index.html** file before proceeding.

- __5. This step will add custom CSS content to main.css file to provide a “branded” look for the EGUI application, overriding defaults and giving a more consistent experience across different devices.

In the Project Explorer view, find and open (**double-click**) **apps/EGUI/common/css/main.css** in the editor, **delete the entire contents** and **copy and paste** the following CSS text. You can also copy and paste code from **EGUI_main.css** from the accompanying Lab Snippets folder. Be sure to **save** the changes to the main.css.

By adding in the CSS Styles provided, we are able to make the application running on all devices look the same. The EGUI/common/css folder contains a stylesheet (main.css) that overrides the CSS for individual platforms (Android, iOS, ect.) so changes we made here will reflect in all device types.

1.5.3 Adding functionality to the views

In this section we will add and update content to main.js JavaScript file in order to provide the logic for the mobile application to function.

- __1. Open the CICSMobile/apps/EGUI/common/js/main.js file. You will see that there are some functions already defined: wlCommonInit() and dojoInit()

```

main.js
function wlCommonInit(){
  require([ "layers/core-web-layer", "layers/mobile-ui-layer" ], dojoInit);

  /*
   * Application is started in offline mode as defined by a connectOnStartup prop
   * In order to begin communicating with Worklight Server you need to either:
   *
   * 1. Change connectOnStartup property in initOptions.js to true.
   *    This will make Worklight framework automatically attempt to connect to W
   *    Keep in mind - this may increase application start-up time.
   *
   * 2. Use WL.Client.connect() API once connectivity to a Worklight Server is re
   *    This API needs to be called only once, before any other WL.Client methods
   *    Don't forget to specify and implement onSuccess and onFailure callback fu
   *
   *    WL.Client.connect({
   *      onSuccess: onConnectSuccess,
   *      onFailure: onConnectFailure
   *    });
   *
   */

  // Common initialization code goes here
}

```

- __2. Rather than manually adding the implementation, you can simply override the content with the main.js file included the lab resources. This can be done in several ways:
 - __a. This can be done by deleting all of the content of the current main.js file then pasting the content from the EGUI_main.js file that is part of the supplied lab resources.
 - __b. Alternatively, this can be done by deleting the main.js file in the common/js folder then copying the EGUI_main.js file into the common/js directory and renaming to main.js
- __3. Review the comments to get a better understanding of what each function provides for the mobile application.

1.6 Connecting to CICS using Worklight Adapter

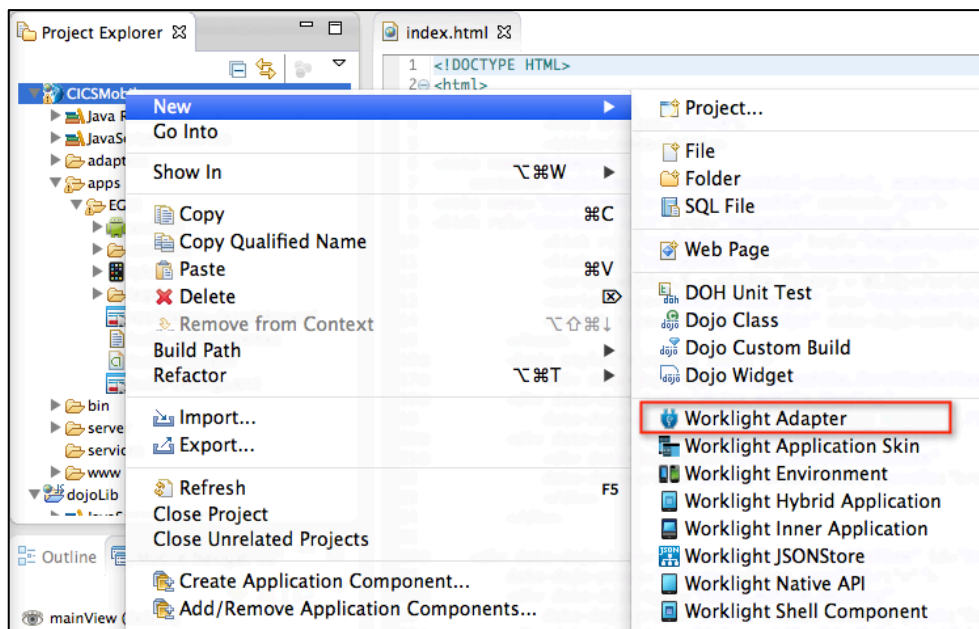
In this portion of the lab, you will create a Worklight HTTP adapter and integrate connect into the CICS EGUI application.

A Worklight adapter is hosted on the Worklight server, and interacts with remote data sources, retrieving data or performing actions. The Worklight client runtime provides a simple, common JavaScript interface to invoke the adapter and exchange data from a mobile application.

In this lab you will be connecting the mobile front-end application that you have built with back-end services and data from the original version of this application, running in CICS Transaction Server on z/OS. This 3270 CICS application has been web-service enabled so that we can use modern application interfaces, such as Web and Mobile, to re-use the business logic and access data from our legacy application. These web-services can be communicated with by sending SOAP messages through HTTP 'post' requests. In this next section, you will build the Worklight Adapter that contain the messaging structure for sending and receiving information to/from the System z.

__1. Create a new Worklight Adapter.

__a. In the Project Explorer view, right-click on **CICSMobile** project > **New > Worklight Adapter**.



__b. In the New Worklight Adapter dialog, select the Adapter type **HTTP Adapter** and enter **CatalogWrapperAdapter** as the name then click the **Finish** button.

Worklight Adapter
Create a new adapter.

Project name: CICSMobile

Adapter type: HTTP Adapter

Adapter name: CatalogWrapperAdapter

Create procedures for offline JSONStore

Retrieve JSON data with:

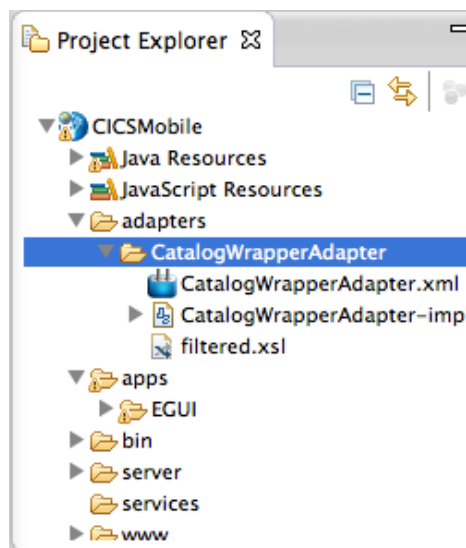
Add JSON data with:

Replace JSON data with:

Remove JSON data with:

Cancel Finish

- __c. The **CatalogWrapperAdapter** Adapter will be created with the following contents



CatalogWrapperAdapter.xml – Adapter configuration containing connection info, security info and registered methods for the adapter

CatalogWrapperAdapter -impl.js – JavaScript implementation file for the adapter methods

filtered.xsl – XSL stylesheet for use in filtering/processing returned data

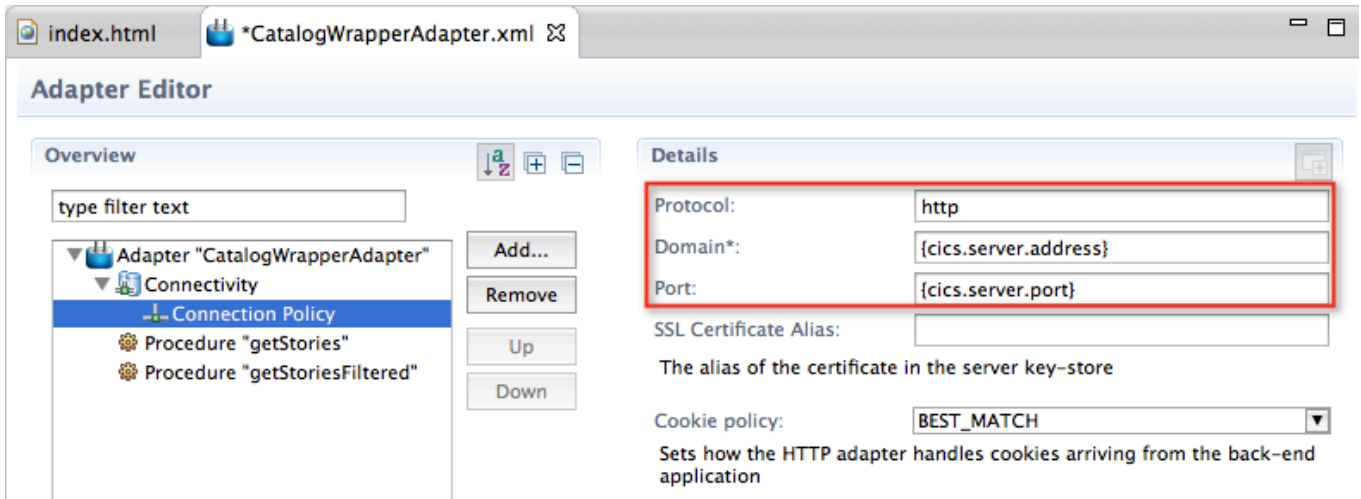
- __2. Edit the CatalogWrapperAdapter xml file

The **CatalogWrapperAdapter.xml** file should be open in the editor, with tabs for Design and Source. The default adapter is created as a sample to retrieve RSS feed data from cnn.com and offers methods for both raw and field-filtered data. We will over-write the default values and methods to connect to our CICS application running on a backend zOS server.

- __a. Select the **Design** tab of the CatalogWrapperAdapter.xml editor (on the bottom left), expand the **Connectivity** node and select the **Connection Policy** to edit the HTTP connection details. The image below shows an example of the values that could be used.

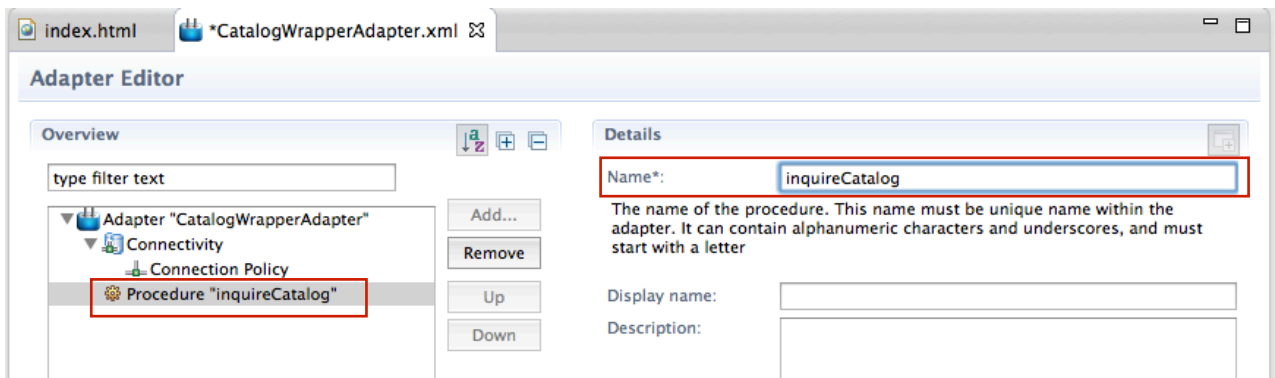
__b. Change the Domain to **{cics.server.address}**

__c. Change the Port to **{cics.server.port}**



__e. Select **Procedure "getStoriesFiltered"** > click the **Remove** button

__f. Select **Procedure "getStories"** and change the name to **inquireCatalog**



__g. **Save** the file.

This step replaces the two procedure entries for `getStories` and `getStoriesFiltered` with a new procedure called **inquireCatalog** (shown in the Source view below).


```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!--
3   Licensed Materials - Property of IBM
4   5725-I43 (C) Copyright IBM Corp. 2011, 2013. All Rights Reserved.
5   US Government Users Restricted Rights - Use, duplication or
6   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
7   -->
8 <wl:adapter name="CatalogWrapperAdapter"
9   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
10  xmlns:wl="http://www.worklight.com/integration"
11  xmlns:http="http://www.worklight.com/integration/http">
12
13  <displayName>CatalogWrapperAdapter</displayName>
14  <description>CatalogWrapperAdapter</description>
15  <connectivity>
16    <connectionPolicy xsi:type="http:HTTPConnectionPolicyType">
17      <protocol>http</protocol>
18      <domain>{cics.server.address}</domain>
19      <!-- Following properties used by adapter's key manager for
20      <sslCertificateAlias></sslCertificateAlias>
21      <sslCertificatePassword></sslCertificatePassword>
22      -->
23      <port>{cics.server.port}</port>
24    </connectionPolicy>
25    <loadConstraints maxConcurrentConnectionsPerNode="2" />
26  </connectivity>
27
28  <procedure name="inquireCatalog" />
29 </wl:adapter>

```

- __3. Add the implementation for the **inquireCatalog** procedure to the CatalogWrapperAdapter-impl.js file.

By default, adapters are implemented in JavaScript. In this step, open the CatalogWrapperAdapter -impl.js file, remove the code for the default procedures provided by the template and paste the code for our **inquireCatalog** procedure.

- __a. Expand **CICSMobile** project > expand **adapters** folder > expand **CatalogWrapperAdapter** folder > **double-click** to open the **CatalogWrapperAdapter -impl.js** file into the editor.
- __b. Remove the code and comments for the 3 functions **getStories**, **getStoriesFiltered**, and **getPath**. You can simply **delete the entire contents** of the **CatalogWrapperAdapter-impl.js** file.
- __c. Paste the following JavaScript into CatalogWrapperAdapter -impl.js from the **Snippet#3.txt** file in the lab resources.

```

function inquireCatalog(obj) {

    var request = "<soapenv:Envelope
xmlns:soapenv='http://schemas.xmlsoap.org/soap/envelope/'
xmlns:exam='http://www.exampleApp.inquireCatalogRequest.com'>" +

```

```

        "<soapenv:Header/>" +
        "<soapenv:Body>" +
            "<exam:inquireCatalogRequest>" +
        "<exam:startItemRef>" + obj.itemRef + "</exam:startItemRef>" +
        "<exam:itemCount>" + obj.itemCount + "</exam:itemCount>" +
            "</exam:inquireCatalogRequest>" +
            "</soapenv:Body>" +
            "</soapenv:Envelope>";

    var input = {
        method : 'post',
        returnedContentType : 'xml',
        path : 'exampleApp/inquireCatalogWrapper',
        body: {
            content: request.toString(),
            contentType: 'text/xml; charset=utf-8'
        }
    };

    return WL.Server.invokeHttp(input);
}

```

- __d. The resulting implementation should look something similar to this:

```

index.html CatalogWrapperAdapter.xml CatalogWrapperAdapter-impl.js
* Licensed Materials - Property of IBM
* 5725-I43 (C) Copyright IBM Corp. 2011, 2013. All Rights Reserved.
* US Government Users Restricted Rights - Use, duplication or
* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
*/

⊕ * WL.Server.invokeHttp(parameters) accepts the following json object as an argument:[]
⊖ function inquireCatalog(obj) {
    var request = "<soapenv:Envelope xmlns:soapenv='http://schemas.xmlsoap.org/soap/envelope/'>
        <soapenv:Header/> +
        <soapenv:Body> +
            <exam:inquireCatalogRequest> +
                <exam:startItemRef>"+obj.itemRef+"</exam:startItemRef> +
                <exam:itemCount>"+obj.itemCount+"</exam:itemCount> +
            </exam:inquireCatalogRequest> +
        </soapenv:Body> +
    </soapenv:Envelope>";

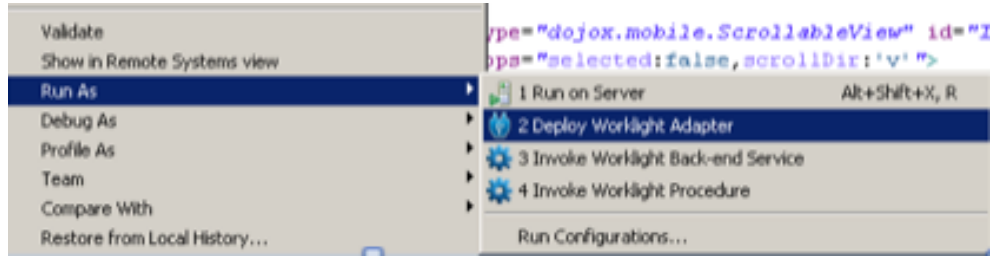
    var input = {
        method : 'post',
        returnedContentType : 'xml',
        path : 'exampleApp/inquireCatalogWrapper',
        body: {
            content: request.toString(),
            contentType: 'text/xml; charset=utf-8'
        }
    };

    return WL.Server.invokeHttp(input);
}

```

As a summary, our adapter configuration file (**CatalogWrapperAdapter.xml**) contains the protocol (**http**), host (**cics.server.address**) and port (**cics.server.port**) information which defines the target System z server address and port that the original 3270 CICS application has been web-service enabled for communication on. Our procedure defines the remainder of the service URL (**/exampleApp/inquireCatalogWrapper**), the HTTP method (**post**) and the content type to expect (**xml**). There are two parameters we are passing into the adapter, as required by the back-end service that we are invoking. These parameters include 'itemRef' which tells what item number we are starting to retrieve data from in the catalog, and 'itemCount' which tells how many items in the catalog that we want data for.

- __4. Deploy the CatalogWrapperAdapter Worklight Adapter.
 - __a. Save all files.
 - __b. Select the **CatalogWrapperAdapter** folder in the Project Explorer, **right-click** and select **Run As > Deploy Worklight Adapter**.

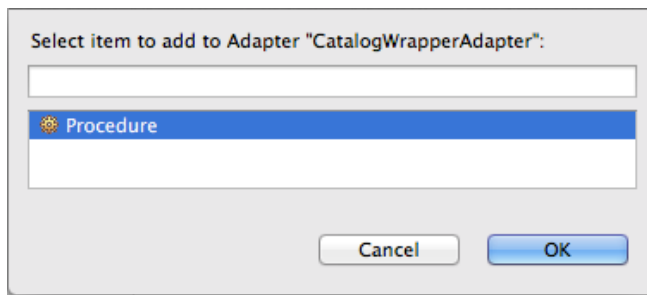


__c. Watch the **console** for the message that the adapter has been successfully deployed.

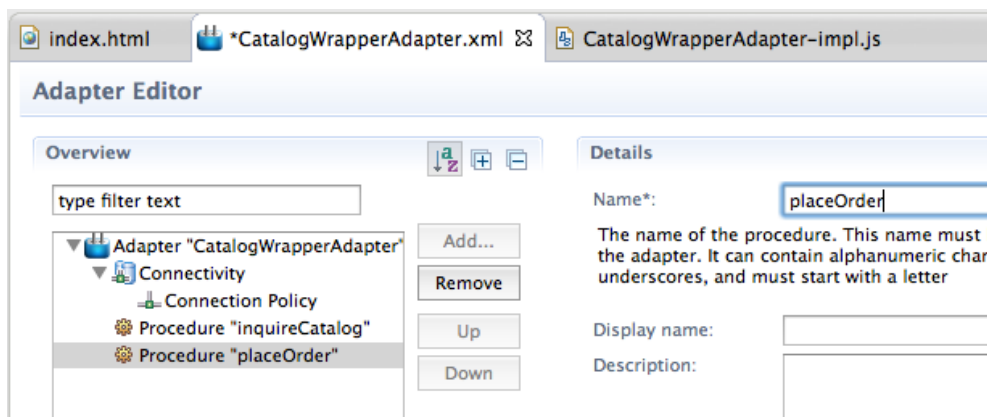
```
[2014-03-10 12:10:50]      Starting adapter deployment on Worklight Server
[2014-03-10 12:10:50]      Starting build of adapter: CatalogWrapperAdapter
[2014-03-10 12:10:50]      Deploying adapter: CatalogWrapperAdapter
[2014-03-10 12:10:50]      Server host: 192.168.11.126
[2014-03-10 12:10:50]      Server port: 10080
[2014-03-10 12:10:51]      Adapter build and deploy finished.
```

__5. Add a second procedure to the CatalogWrapperAdapter Worklight Adapter.

__a. In the **Design** view of the CatalogWrapperAdapter.xml select the CatalogWrapperAdapter name then click on the **Add** button. A dialog will open, select Procedure then click **OK**.



__b. Enter **placeOrder** as the name of the procedure.



__c. Save the CatalogWrapperAdapter.xml file.

- __6. Add the implementation for the **placeOrder** procedure to the CatalogWrapperAdapter -impl.js file by copy and pasting the following JavaScript from Snippet#4.txt into CatalogWrapperAdapter-impl.js after the inquireCatalog() function.

```
function placeOrder(obj) {

    var request = "<soapenv:Envelope
xmlns:soapenv='http://schemas.xmlsoap.org/soap/envelope/'
xmlns:exam='http://www.exampleApp.placeOrderRequest.com'>" +
        "<soapenv:Header/>" +
        "<soapenv:Body>" +
            "<exam:placeOrderRequest>" +
                "<exam:orderRequest>" +
                    "<exam:userId>"+obj.userId+"</exam:userId>"
+
        "<exam:chargeDepartment>"+obj.dept+"</exam:chargeDepartment>" +
        "<exam:itemReference>"+obj.itemRef+"</exam:itemReference>" +
        "<exam:quantityRequired>"+obj.quantity+"</exam:quantityRequired>" +
            "</exam:orderRequest>" +
            "</exam:placeOrderRequest>" +
            "</soapenv:Body>" +
            "</soapenv:Envelope>";

    var input = {
        method : 'post',
        returnedContentType : 'xml',
        path : 'exampleApp/PlaceOrderWrapper',
        body: {
            content: request.toString(),
            contentType: 'text/xml; charset=utf-8'
        }
    };

    return WL.Server.invokeHttp(input);
}
```

- __7. The resulting implementation should look something like this:

```

index.html CatalogWrapperAdapter.xml CatalogWrapperAdapter-impl.js
* WL.Server.invokeHttp(parameters) accepts the following json object as an argument:
function inquireCatalog(obj) {}
function placeOrder(obj) {
    var request = "<soapenv:Envelope xmlns:soapenv='http://schemas.xmlsoap.org/soap/envelope/' xmlns
                  <soapenv:Header/>" +
                  "<soapenv:Body>" +
                    "<exam:placeOrderRequest>" +
                      "<exam:orderRequest>" +
                        "<exam:userId>"+obj.userId+"</exam:userId>" +
                        "<exam:chargeDepartment>"+obj.dept+"</exam:chargeDepartment>" +
                        "<exam:itemReference>"+obj.itemRef+"</exam:itemReference>" +
                        "<exam:quantityRequired>"+obj.quantity+"</exam:quantityRequired>" +
                      "</exam:orderRequest>" +
                    "</exam:placeOrderRequest>" +
                  "</soapenv:Body>" +
                "</soapenv:Envelope>";

    var input = {
        method : 'post',
        returnedContentType : 'xml',
        path : 'exampleApp/placeOrderWrapper',
        body: {
            content: request.toString(),
            contentType: 'text/xml; charset=utf-8'
        }
    };

    return WL.Server.invokeHttp(input);
}

```

In summary, we added a second procedure call to the existing adapter because we are going back to the same CICS region. Our procedure defines the remainder of the service URL (**/exampleApp/placeOrderWrapper**), the HTTP method (**post**) and the content type to expect (**xml**). There are four parameters we are passing into the adapter, as required by the back-end service that we are invoking. These parameters include **'itemRef'** which tells what item number we are placing an order for, **'quantity'** which tells how many of the item we want from the catalog, **'userId'** which is the I.D. of the user placing the order, and **'dept'** which is the code for the department that will be charged for the order.

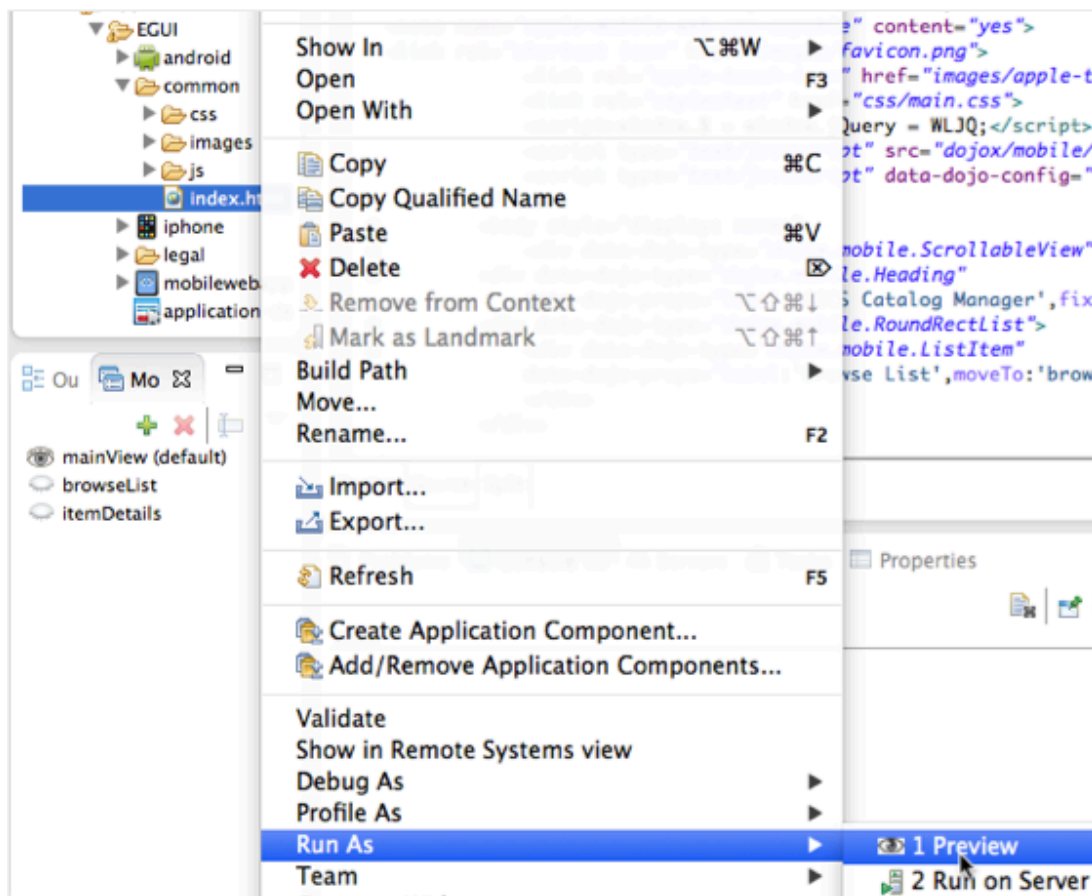
- __8. Deploy the updated CatalogWrapperAdapter.
 - __a. Save all files.
 - __b. Select the **CatalogWrapperAdapter** folder in the Project Explorer, **right-click** and select **Run As > Deploy Worklight Adapter**.
 - __c. Watch the **console** for the message that the adapter has been successfully deployed. This is similar to the result the first time the adapter was deployed.

Congratulations you have successfully used the Worklight Studio to build a Worklight application with multiple mobile device environments, navigation between views, and are set to retrieve data from a back-end service with a Worklight adapter.

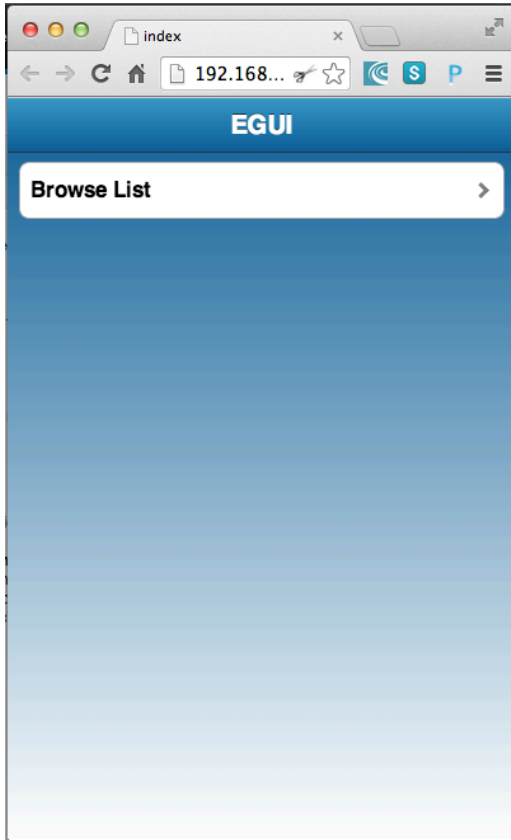
1.7 Test with Mobile Browser Simulator

The final step is to test the EGUI application using the Mobile Browser Simulator (MBS). A new external browser window will open with the simulator rendering our EGUI application. In order to highlight the capabilities of the MBS, we will start with a basic web preview first.

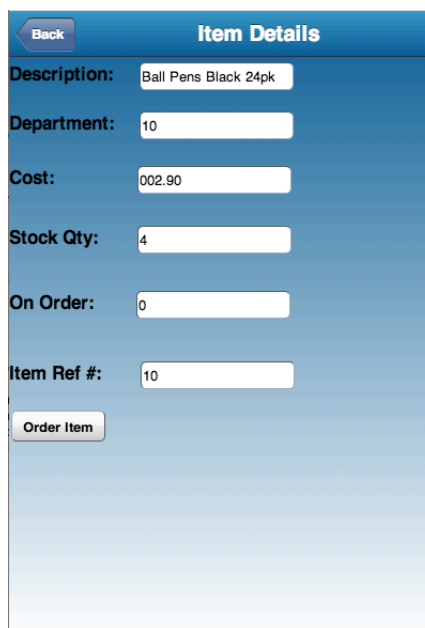
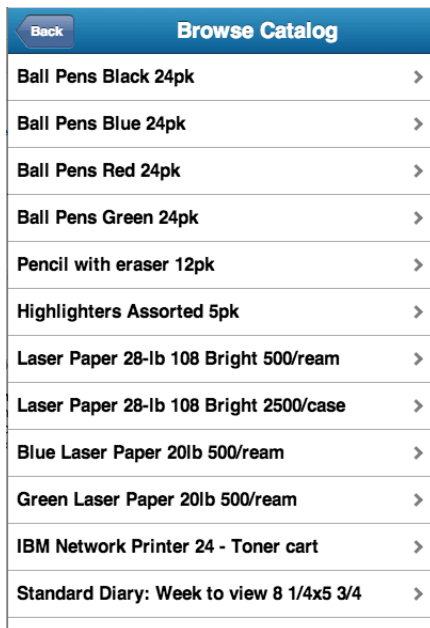
- __1. This Step will use the basic web preview to look at the mobile application using a plain browser.
 - __a. Right click on the **common** folder (to run the common resource web app) or the index.html file within the common folder and select **Run as > Preview** (as shown below). This opens the mobile application as basic web page.



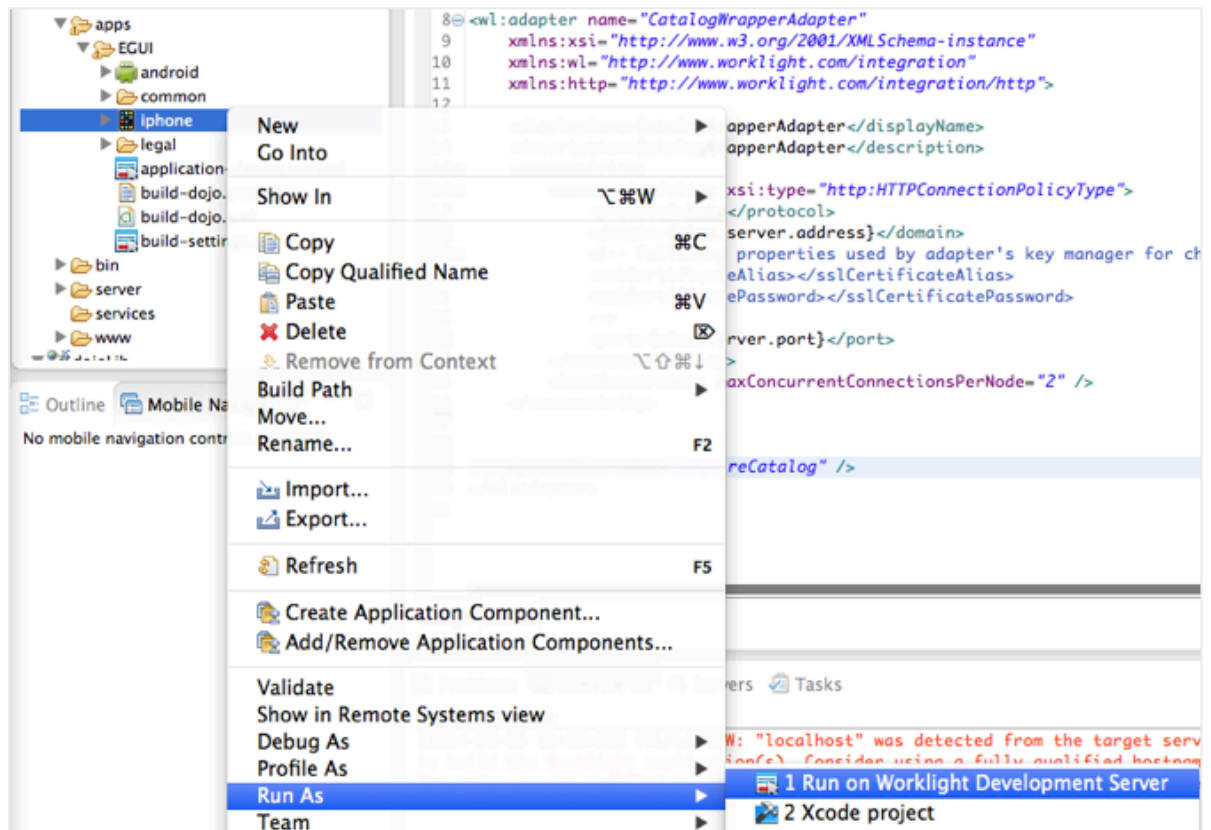
- __b. A new external browser window will open in which the EGUI application should be displayed. (This may take a few seconds to a few minutes while the application is built and deployed completely). The browser window will open into a full screen. Resize the window to a smaller size, more fitting of a mobile device for demonstration purposes. We can see how MBS does this in later steps.



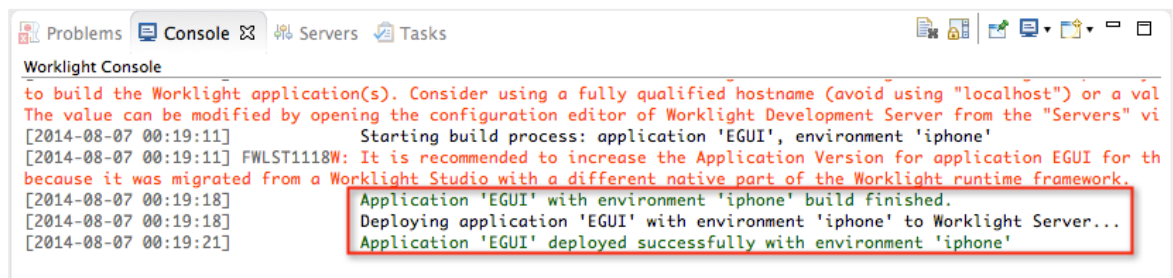
- c. Navigate the views to validate that the adapter is working and data is being retrieved. After you place a successful order, you can check the item stock quantity to validate that it was indeed decreased by 1 (default setting for this lab). The implementation can be found in the main.js.



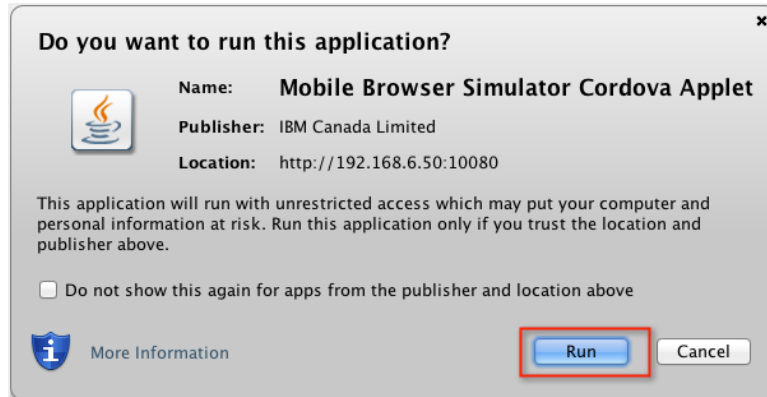
- __d. After successfully placing an order, refresh the web page to return the main screen. Browse the catalog and select the item that was order previously to validate that the Stock Qty has been depleted.
- __2. This step will now show what the test preview looks like in the Mobile Browser Simulator.
- __a. Right click on either the **iphone** or **android** (depends on your preference) folder and select **Run as > Run on Worklight Development Server** (as shown below). This ensures that the application environment is deployed to server, if not already.



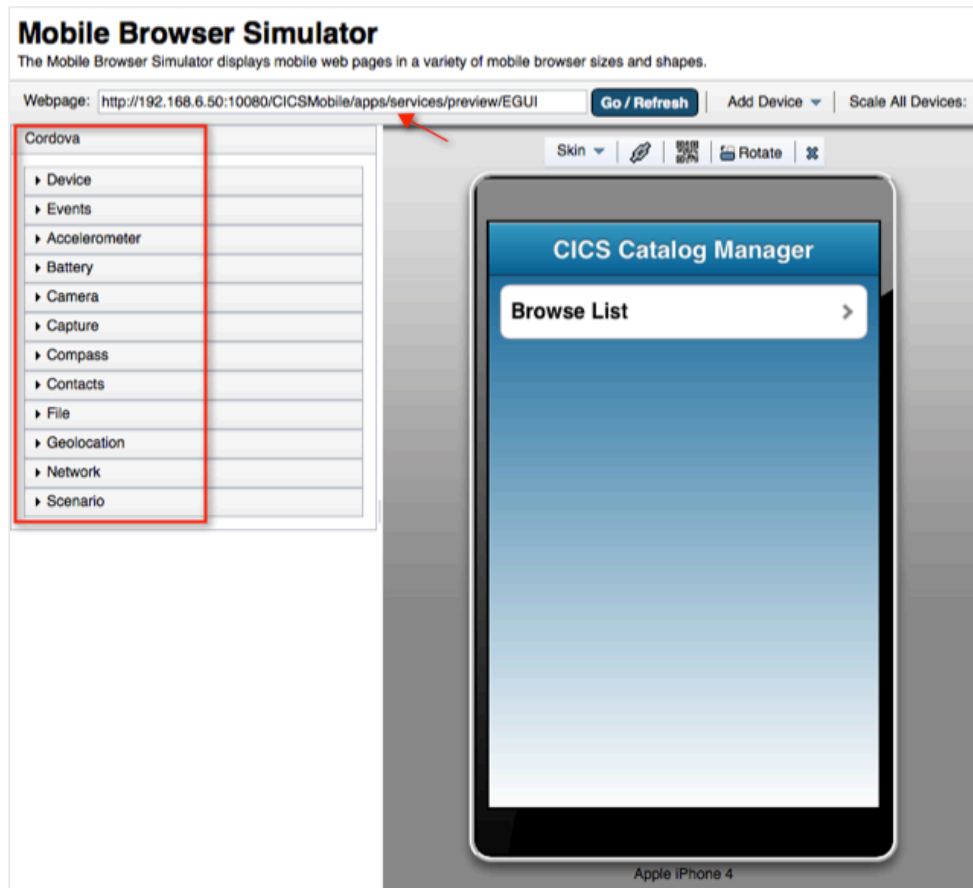
- __b. Wait for the Worklight Console to display that the deployment is successful, then proceed.



- c. Right click on either the **iphone** or **android** (based on the previous selection) folder and select **Run as > Preview**. This will open a new browser (or tab), this time you will see the Mobile Browser Simulator look and feel. If prompted, please make sure to allow Java applet or plugin to run.



- d. After successful initialization, you should see something similar to the image below. Notice that it is still a browser url address being accessed, however there are now different device sensors and features available to test. Even though the current state of the application does not exploit any device feature yet. If there are features like camera calls implemented, it can be tested by using these simulated API calls.




1.8 Summary

In this lab you have learned how to use Worklight Studio to create a cross platform mobile application using HTML5, CSS, JavaScript, the Dojo Mobile framework. You have learned how to target specific devices such as iPhone and Android phones and seen some very basic customizations that can be applied for making the application look consistent across platforms. You have also seen how to use the Mobile Browser Simulator within both the Worklight Studio and the Worklight Console to test the application in preview mode. Feel free to explore the different options available in the simulator.

You should now be familiar now with the Worklight Studio environment and the common development tasks associated with building a mobile application extending the capabilities of existing enterprise applications running on System z.

Congratulations!



MOBILE MAINFRAME APP THROWDOWN

Will you be our mobile champ?

CICS | IMS | WAS | DB2

Open to existing System z clients

The challenge: Build a proof-of-concept demonstrating mobile enablement of your existing mainframe apps.

Get IBM help to build your mobile PoC

Call us 'Coach': We provide getting started guides and access to IBM zMobile Experts for questions and queries.

Win a week with IBM experts & more

Make it real: Win help from IBM to bring your mobile app to life.

ibm.biz/mmathrowdown No submission of code required, only screenshots.
Entries must be complete and submitted by **17 Sept 2014**.