A Cross-Platform “Hello World!” for Hybrid

In this lab, we will show you how to build a basic hybrid app for iOS and Android, sharing the web portion of the codebase between the iOS and Android projects.

1. Create a Zurb Foundation web application
   1. Make a directory in your working sandbox folder titled "HelloWorldXPlatform"
   2. Open terminal
   3. Navigate to the folder you created in the first step (ex. "cd ~/Projects/[your repo]/HelloWorldXPlatform")
   4. Use the Zurb Foundation framework foundation command to create web app:

foundation new web

This will kick a pull of the latest Zurb Foundation boilerplate and drop it in a folder named web.

1. Modify Zurb Foundation boilerplate - Add Zepto
   1. Open a Finder window and navigate to Projects/jslibraries/zepto-dist
   2. Select the zepto.min.js library and copy it (command + C)
   3. Launch WebStorm
   4. Open the web directory you created above
   5. Select the js folder in the Project view, and paste the file you copied (command + V). Click on OK at the prompt.

1. Modify Zurb Foundation boilerplate - Replace index.html
   1. We're not going to use the index.html page it created. It's just a bunch of examples of what you can do with Foundation. So we're going to replace that with our own index.html.
   2. Delete the index.html it created
   3. Right-click on web folder, and from the context menu, and select New > Foundation HTML5 with Zepto. You'll be prompted to give it a name. Name it "index.html". If the template isn’t present, create a new generic HTML file by selecting New > HTML and replace the HTML code it creates with the following snippet:

<!DOCTYPE html>

<html class="no-js" lang="en">

<head>

<meta charset="utf-8"/>

<meta name="viewport" content="width=device-width, initial-scale=1.0"/>

<title></title>

</head>

<body>

<div class="row">

<div>

<h1></h1>

</div>

</div>

</body>

</html>

1. Fix Path to Foundation SASS @import

Because of the way the bower package is installed, we’re losing our reference to it in the default .scss file, so we need to tweak it a little. Change the imports at the top of the file to read as follows, or enter them if they aren’t there already:

@import “settings”;

@import “../bower\_components/foundation/scss/foundation”;

1. Modify Zurb Foundation boilerplate - Edit SASS config.rb
   1. Open the config.rb file at the root of the web folder, or create a new file, and name it config.rb if it does not already exist.
   2. change line 9:

javascripts\_dir = "javascripts"

to

javascripts\_dir = "js"

* 1. If the file didn’t exist there previously, add the following block to your newly created config.rb file:

# Require any additional compass plugins here.

add\_import\_path "bower\_components/foundation/scss"

# Set this to the root of your project when deployed:

http\_path = "/"

css\_dir = "stylesheets"

sass\_dir = "scss"

images\_dir = "images"

javascripts\_dir = "js"

# You can select your preferred output style here (can be overridden via the command line):

# output\_style = :expanded or :nested or :compact or :compressed

# To enable relative paths to assets via compass helper functions. Uncomment:

# relative\_assets = true

# To disable debugging comments that display the original location of your selectors. Uncomment:

# line\_comments = false

# If you prefer the indented syntax, you might want to regenerate this

# project again passing --syntax sass, or you can uncomment this:

# preferred\_syntax = :sass

# and then run:

# sass-convert -R --from scss --to sass sass scss && rm -rf sass && mv scss sass

1. Turn on WebStorm Compass Support
   1. Go to WebStorm > Preferences. In the dialog that pops up, enter “Compass” in the search window. It should return a section titled “Compass Support”. Click on enable. When you do, the “Compass executable file” and “Config path” fields should be automagically populated if they haven’t been already. If they are not, see your Instructor.

1. Modify Zurb Foundation boilerplate - Turn on LiveEdit in WebStorm
   1. From the WebStorm menu, click on View and select LiveEdit. You should now see a checkmark to the left of it.
   2. Time to have some fun. Right-Click on the index.html file you created. In the context menu, select "Debug 'index.html'". This will launch Chrome and load the WebStorm debug extension, and will then load your index.html page. It's blank right now, but we're about to change that.

1. Build index.html page
   1. Arrange WebStorm and the Chrome instance running your index.html page so that they are open side-by-side.
   2. With your Chrome window still open and in WebStorm debug mode, Open your index.html page in WebStorm if it isn't open already. In the first header tags you see around line 12, enter "Hello World!" between those tags, and watch what happens to your page in chrome as you type. It's updating as you type! If you’re not seeing that magic, you may need to shutdown and re-run Debug again to give it the old Fonzie hit and get it going. The fun ain't done yet! We're just getting started!
   3. We're going to use some of Foundation's SASS frameworks to apply some nice styling to our page. In the div that wraps the h1 tag you just edited, add the following CSS class attributes:

class="large-12 columns text-center"

Note that your "Hello World!" text is now center-justified in Chrome.

* 1. Time to add a button that we're going to use for a few things later. After the "row" div that wraps your "Hello World!" header, we're going to add a new Foundation "row" div. Insert the following snippet AFTER the closing div tag for the first "row":

<div class="row">

<div class="large-12 columns text-center">

<a href="http://www.google.com" class="small radius button">Make Some Magick!</a>

</div>

</div>

...And we get some more LiveEdit magic. Note that a beautiful blue button just magically appeared right below your "Hello World!" header. Now we're going to add a little styling of our own.

1. Style index.html page
   1. Open your app.scss file and add the following chunk of SASS to it:

$backgroundColor: #C0C0C0;

body {

background-color: $backgroundColor;

div:first-of-type {

margin-top: 20px;

}

}

Compass will pick these changes up, recompile the app.css file, and then reload your index.html page, which should now have a grey background. Note that we're using SASS variables to set the color value ($backgroundColor: #C0C0C0;). That way, we've got a one-stop shop for tweaking this particular color, and it can be reused anywhere.

* 1. Add another variable to your app.scss file, directly below the $backgroundColor variable:

$buttonColor: #3c9a5f;

and below the body styling you created, add the following:

.button-green {

background-color: $buttonColor;

}

But nothing is changing on the page yet. That's because we have to add the style to our button. Go back to your index.html and add "button-green" to your anchor:

<a href="http://www.google.com" class="small radius button button-green">Make Some Magick!</a>

Note that your button has now changed to a pleasant shade of green.

* 1. Just a couple more things to do and we're golden. Add the following CSS to your app.scss file:

.row:first-of-type {

margin-top: 20px;

}

.bottom-justify {

position: fixed;

bottom: 1px;

}

Note that your "Hello World!" header just dropped down a bit. We needed to do that to make room for the status bar on iOS 7 devices.

Now we're going to make use of that bottom-justify class to do just that with the button we built. In the Foundation "row" div that the button is wrapped in, add our bottom-justify class after "row", so that the whole block now looks like:

<div class="row bottom-justify">

<div class="large-12 columns text-center">

<a href="http://www.google.com" class="small radius button button-green">Make Some Magick!</a><br/>

</div>

</div>

Note now that the button is now at the bottom of the screen, and it remains sticky in relation to the bottom edge of the browser window.

1. Test the button

You'll recall that our button is just an anchor linked to Google. Click on it, and should navigate to google.com. We're going to do something special with this button a little later. For now, the HTML and CSS parts of your web are all done!

1. Build the iOS container
   1. Fire up Xcode
   2. From the Xcode menu, go to File > New Project. Xcode will launch a wizard with some template options
   3. The template selected by default should be "Single View Application". Use that template and click next.
   4. For the next screen, under Product Name, name it "HelloWorld". Leave the Class Prefix field empty. For devices, if you are using an iPod Touch or iPhone, choose iPhone; if you are using an iPad, select iPad. Leave everything else with their defaults and click next.
   5. You're now going to be prompted for a project location. In the file explorer window, navigate to the top of your HelloWorldXPlatform folder, where you should see only the web folder you created in the steps above. Do\_not create the project in the web folder. We want to create it at the same level as the web folder, because we're going to be sharing this code between the iOS project you are creating now, and the Android project you'll be creating in a bit. Click on the Create button. Xcode will create a root folder and generate a project within it from the template.
   6. We need to add the Zurb Foundation web app we created earlier now. Go to the Project Explorer (left-hand pane) and right-click on the HelloWorld folder at the top, directly underneath the project file parent node. From the context menu that appears, select "Add files to "HelloWorld"". This will open a file dialog. Using the folder drop-down at the top of the dialog, navigate up the heirarchy two levels, to your "HelloWorldXPlatform" folder. Select the web folder. Add the folder to the project with the following settings (these may be setup by default, but just to be sure):

Destination: Copy items into destination group's folder (if needed) - unchecked

Folders: Select "Create folder references for any added folders" radio button

Add to Targets: Leave default "HelloWorld" target checked. Don't check the "HelloWorldTests" target

Click on the Add button. You may be prompted to "Share Working Copy". If so, confirm that this is okay.

* 1. From the Project Explorer in the left-hand pane, select the Main.storyboard file. You should now see the interface builder in the main view.
  2. We're going to add a UIWebView as a child of the UIView that is in there by default. To do this, click on the search box in the bottom right-hand corner of Xcode. This is a search tool for the various libraries that come with Xcode. The object libary should be selected by default. The way to tell is the cube icon at the top of the pane will be highlighted. Do a search for "webview". You should now see the Web View (UIWebView) in the search results. Click and drag that over the main UI view, and let go. In the Document Outline View that appears between the Project Explorer and the UI Designer, you should now see that you have a Web View that is a child of the View that was created by the template. Drag the view so that is positioned at 0,0, and fills the Main UI view. Note that Interface builder is giving you some guides with visual cues that show you when you've got your UIWebView centered and that it is filling the screen.
  3. We need to set some default properties on the WebView now. With the Web View selected in the Document Outline, click on the Super Hero belt buckle at the top of the right hand pane, and check the following options in the "Web View" section that appears directly below it:

* + - Scales Page To Fit
    - Links
    - Addresses
    - Events

Leave the rest of the defaults as is.

* 1. j. Because we've made our Web View a child of a UIView, it won't dynamically resize to the dimensions of the parent unless we instruct it to do so either at design time or at runtime. As a result, it will always load and retain the height and width dimensions that were setup in Interface Builder, which means that it won't resize itself as needed for different iPhone screen sizes. We're about to remedy that with a newer feature to iOS called AutoLayout. In the bottom right-hand corner of Interface Builder's main area, there are 3 toolboxes. The middle one has 4 buttons in it. To add an AutoLayout Constraint, With the Web View still selected in the Document Outline pane, we're going to click on the 2nd one, which looks sorta like |- + -|. That will pop an AutoLayout dialog. For each of the 4 dropdowns in the top section of the dialog, select "Use Current Canvas Value". Note that, when done, the white box in the middle now has a highlighted red "beam" on each side. Click on the "Add 4 Constraints" button at the bottom of the dialog. Note that now in the Document Outline pane, there is a "Constraints" box at the same level as the Web View. The Web View will now always dynamically resize to the dimensions of the parent view.
  2. We need to wire our UIWebView up to the ViewController now. Open ViewController.h and add the following property to it:

@property (nonatomic, strong) IBOutlet UIWebView\* webView;

Now go back to the Main.storyboard file, and this time, select the "View Controller" node in the Document Outline pane. In the Inspector pane at the far right, click on the right-turn arrow icon at the top. You should now see 5 sections below: Outlets, Outlet Collections, Referencing Outlets, Referencing Outlet Collections, and Received Actions. In the Outlets section, you should see the webView property you added in the step above.

* 1. Roll over the little circle to the write of the webView outlet in that section. It will turn into a plus sign. When it does, click-hold and drag over the main UI view in Interface builder. A "wire" will follow your click/drag, and then your Web View will be highlighted in blue, with a little "Web View" box displayed in the bottom left hand corner. Let go of the mouse button as soon as you see this, and your IBOutlet property will now be wired up to your UIWebView. You can confirm this by the change that occurred with your webView property in the Outlets section of the left-hand pane. Note that it now displays two balloons, "webView" and "Web View", with a connector between them.

* 1. Time to write some Objective-C! From the Project Explorer, navigate to the ViewController.m file. In the ViewController's viewDidLoad method, add the following code after the [super viewDidLoad] call and comment on the following line:

self.webView.scrollView.showsHorizontalScrollIndicator = NO;

self.webView.scrollView.showsVerticalScrollIndicator = NO;

NSString\* htmlPath = [[NSBundle mainBundle] pathForResource:@"web/index" ofType:@"html"];

[self.webView loadRequest:[NSURLRequest requestWithURL:[NSURL fileURLWithPath:htmlPath isDirectory:NO]]];

Set your project to Run from one of the simulators if it isn't already. If you click anywhere on the device target that is displayed after the project target in Xcode's upper left-hand corner (after the hierarchy arrow that appears next to the Project Target), You'll get a drop-down with a list of simulator options, and your iOS device if it is connected. Select one of the simulators specific to the device type you setup for your project (iPhone simulator if you created an iPhone project, iPad simulator if you selected an iPad project), and click on the "Play" button in Xcode's upper left-hand corner. Xcode will build the project, deploy it to the simulator, launch the simulator and run your app in bug mode. Voila! Hello World for iOS! If your device has been provisioned for development, you should be able to run it there as well. Also, try tapping on your button. It will navigate to Google.com on click, all within your web view.

Hang tight. We still have a ways to go.

1. Build a Javascript to Native Bridge for iOS and Intercept the google.com link
   1. It's easier than it sounds. Really. We're going to redirect the google.com link in your button into your native container, and then we're going to use that click action to display a native "Hello World" label that will lay over the top of your web view. First, let's work on intercepting the HTTP GET to google.com that is happening in your code now. First, we need to add the UIWebView's delegate to the ViewController header file. Navigate to the ViewController.h file in the Project Explorer. Add <UIWebViewDelegate> to the end of the @interface declaration on line 11 so that it now looks like:

@interface ViewController : UIViewController <UIWebViewDelegate>

We need to override one of the methods on that delegate now to get hooks into the request coming off of the button click. First, Navigate to the ViewController.m file and insert the following line of code after the commment that appears underneath the [super viewDidLoad] call in the - (void)viewDidLoad method:

self.webView.delegate = self;

Now add the following UIWebViewDelegate method override to the bottom of the ViewController class, BEFORE the @end line:

#pragma mark - Optional UIWebViewDelegate delegate methods

- (BOOL)webView:(UIWebView \*)webView shouldStartLoadWithRequest:(NSURLRequest \*)request navigationType:(UIWebViewNavigationType)navigationType

{

NSString\* requestAbsoluteString = [[request URL] absoluteString];

NSLog(@"request: %@", requestAbsoluteString);

if([requestAbsoluteString rangeOfString:@"index"].location != NSNotFound)

return YES;

NSLog(@"Request intercepted! Not going to google!");

return NO;

}

Now try running the app and tapping on your button. It no longer navigates to google.com. Instead, you should now see some log messages in the console that look something like the following:

2013-12-16 17:47:33.233 HelloWorldRunThrough[50844:70b] request: file:///Users/deemadden/Library/Application%20Support/iPhone%20Simulator/7.0.3/Applications/2397C4A1-5A91-4B03-BE8F-41DB5B21E14D/HelloWorldRunThrough.app/web/index.html

2013-12-16 17:47:37.621 HelloWorldRunThrough[50844:70b] request: http://www.google.com/

2013-12-16 17:47:37.621 HelloWorldRunThrough[50844:70b] Request intercepted! Not going to google!

So what's going on here? By attaching to the UIWebView's delegate and overriding the method above, we're now able to watch the HTTP traffic and do something different with it if we choose to do so. By adding the lines:

if([requestAbsoluteString rangeOfString:@"index"].location != NSNotFound)

return YES;

NSLog(@"Request intercepted! Not going to google!");

return NO;

We're saying "for any request other than those that contain 'index' in their URL path, we want to block them from loading". So when you tap on your button and the HTML attempts to navigate to google.com, the URL string doesn't contain "index" anywhere in our criteria, so the request stays in the native container.

Now we're going to do something special with it.

1. Respond to request intercept by displaying a native iOS Label
   1. Navigate to your Main.storyboard file. In the Object Library search box you used earlier, do a search for "label". A label control should turn up. Drag it on to your UI and try to center it in the view. In the Document Outline pane, you should now see a Label at the same level as the Web View, underneath the parent UIView.
   2. With your newly added label still selected, make the following changes to its properties (under the super hero belt buckle in the inspector at the right):

Font: System 20.0

Alignment: Center

Lines: 2

Alpha: 0

* 1. Now navigate to the Size Inspector (ruler to the right of the super hero belt buckle). Enter the following height and width values:

Width: 177

Height: 65

This will have made the label off-center. Move it around until it is centered in the view again.

In the properties inspector (aka super hero belt buckle), Change the text to “Hello from the Native Container!”

* 1. Time to wire it up. Add the following Outlet property to your ViewController header file:

@property (nonatomic, weak) IBOutlet UILabel\* helloWorldLabel;

Now repeat the steps performed above to wire up your webView, this time doing it for the label you added.

* 1. Add the following code to your UIWebView delegate method override, after the "Request intercepted!" log message:

self.helloWorldLabel.text = @"Hello World! From The Native Layer";

if(self.helloWorldLabel.alpha == 0)

{

[UIView animateWithDuration:0.5

delay:0

options:UIViewAnimationOptionCurveEaseIn

animations:^{ self.helloWorldLabel.alpha = 1; }

completion:nil];

}

else

{

self.helloWorldLabel.textColor = [UIColor blueColor];

}

* 1. Run your app and try clicking on the button now. Watch what happens! On button tap, you should now see the label fade into view with the message "Hello World! From The Native Layer".

1. Build a Native to Javascript command for iOS
2. Jump back into WebStorm and open up your index.html file. Embed the following javascript in a script tag, right before the closing body tag:

<script type="application/javascript">

function helloFromNativeCommand()

{

$('h1').text('Hello World From Native Button!');

}

</script>

1. Jump back into Xcode and add a native button to execute the javascript. Go back to the object library search box and type "button". A button control should appear at the top of the search results. Drag it to the horizontal center of the view, and try to position somewhere halfway between the label you created and the HTML button, which you're not going to be able to see.

1. Add the properties for the button under the super hero belt buckle as follows:

Text = Change the Title

Font = System 19.0

Set the width and height to the following in the size inspector (ruler icon)

Width = 148

Height = 30

Then re-center the label by dragging it around until the guides show you that it is centered.

1. Wire it up. Navigate to the ViewController .h file and add the following property and action:

@property (nonatomic, weak) IBOutlet UIButton\* helloWorldButton;

-(IBAction)helloWorldButtonTapped:(UIButton\*)sender;

And add the following method to the bottom of the ViewController.m file (again, BEFORE the @end tag)

-(IBAction)helloWorldButtonTapped:(UIButton\*)sender

{

}

1. Navigate to the Main.storyboard and select the View Controller in the Document Outline, then click on the right-hand turn signal in the Inspector pane at the far right. This one is a button so the connectors are a little different.

1. Drag the helloWorldButton connector from the Outlets to the button on the UI.
2. Drag the helloWorldButtonTapped: connector from the Received Actions to the UI. When you let go of the button, a context menu will pop. When it does, select "Touch Up Inside". You should be all hooked up now.
3. We can execute the helloFromNativeCommand() javascript function we embedded in our index.html in one line of code. Add the following line to the -(IBAction)helloWorldButtonTapped:(UIButton\*)sender method:

[self.webView stringByEvaluatingJavaScriptFromString:@"helloFromNativeCommand()"];

the secret sauce is UIWebView's stringByEvaluatingJavaScriptFromString method. It allows you to send messages or commands to you javascript layer.

1. Run the app and tap on your "Change the Title" button. The "Hello World!" header in your html will change to "Hello World From Native Button!" on tap! You've built your first javascript bridge!

1. Build an Android container
   1. Now we're going to build a similar container for Android. Fire up Android Studio. In the dialog that appears, click on "New Project..." from the "Quick Start" pane.
   2. Add the following to the New Project Form:

Application name = Hello World App

Module name = Leave the default

Package name = com.softsource.helloworldandroidapp

Project location = set it to your root HelloWorldXPlatform folder, and add HelloWorldAndroid (which will be the top-level project folder name) to the end of the path.

Minimum required SDK, Target SDK, Compile with – Leave Defaults. If you have an android device and would like to try deploying to it, see the instructor for help with getting that going.

* 1. Click next.
  2. Leave the icons settings in the next screen on their defaults and click Next.
  3. Leave the activity set to "Blank Activity" and click Next.
  4. Leave the Activity Name, Layout Name, Fragment Layout Name, and Navigation Type all set to their defaults and click Finish. Android studio will build out your project with activity\_main.xml - the file that defines the UI - and a preview of the UI by default.
  5. We need to add the web app to the project now, and unfortunately, it isn't quite as straightforward as what we got to do in Xcode. There are several other ways to do this that are probably more appropriate in the context of a real project, but in the interest of this here "Hello World", we're going to do the simplest thing possible. Right-click on the main directory in the Project Explorer, and select New > Directory. Name your new directory "assets".
  6. From the Android Studio menu, go to Tools > Open Terminal... This will open up a BASH window inside\_of the IDE, at the bottom of it, with the default location at the top of the project folder.
  7. In the terminal, type

mkdir assets

cd HelloWorldRunThroughApp/src/main/assets

to create and navigate into the assets folder.

* 1. To "share" your web application between your Android project and your iOS project, we're going to create a symlink to the web folder in the assets folder you just created. To do this, enter the following in the terminal window:

ln -s ../../../../../web

* 1. Now if you collapse/expand your main folder to refresh, you will see that your web folder is now available as a resource to the project in the assets folder.

* 1. We're going to add an Android WebView to our project now. Open up the activity\_main.xml again under src/main/res/layout. Replace the XML in there with the following code:

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="fill\_parent"

android:layout\_height="fill\_parent"

android:orientation="vertical">

<RelativeLayout

xmlns:tools="http://schemas.android.com/tools"

android:id="@+id/container"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:layout\_alignParentTop="true"

tools:context="com.example.helloworldapp.app.MainActivity"

tools:ignore="MergeRootFrame">

<WebView

android:id="@+id/webview"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent" />

</RelativeLayout>

</RelativeLayout>

You'll get some nags from Android Studio about the parent RelativeLayout element being redundant. Ignore that for now.

* 1. Open the MainActivity class inside of the com.softsource.helloworldandroidapp package. In Androidland, MainActivity is their version of initial ViewController in iOS. Add the following private member directly below the "public class MainActivity extends Activity {" line:

private WebView webView;

And In that class's onCreate method, after the setContentView(R.layout.activity\_main) call, insert the following below the code that is there:

webView = (WebView)findViewById(R.id.webview);

WebSettings webSettings = webView.getSettings();

webSettings.setJavaScriptEnabled(true);

webView.loadUrl("file:///android\_asset/web/index.html");

You will get some nags by Android Studio to add references. By hitting alt+Enter on each one, the dependencies will be added to the project and class.

Try firing it up by clicking on the green bug at the top/center of the IDE. You will be prompted to pick a device or sim to run it in. Choose the HaxmPhone from the "Android virtual device" drop-down at the bottom of the dialog and click on OK. The Android Emulator will start up and load the HaxmPhone sim. Once loaded, you'll be met with a lock screen. Click/drag the lock off screen, and the sim will continue on to your app and load it. But notice now that we're back to square one with our web app. Clicking on the button in your HTML will jump out of the app context, launch Chrome for Android and load google.com. We're about to fix that like we did last time, this time in Java.

1. Add Javascript to Native bridge for Android
   1. We need to add a TextView to activity\_main.xml, after the WebView's RelativeLayout closing tag:

<RelativeLayout android:layout\_width="fill\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical"

android:layout\_alignParentBottom="true">

<TextView

android:id="@+id/helloworldlabel"

android:text="@string/hello\_world\_label"

android:textSize="20sp"

android:layout\_width="175dp"

android:layout\_height="fill\_parent"

android:ellipsize="none"

android:maxLines="100"

android:scrollHorizontally="false"

android:gravity="center"

android:layout\_centerHorizontal="true"

android:visibility="invisible"/>

</RelativeLayout>

And the following string definition to the strings.xml file in src/main/res/values:

<string name="hello\_world\_label">Hello World! From The Native Layer</string>

Then add the following code to the MainActivity's onCreate method, BEFORE that last webView.loadUrl line:

TextView helloWorldLabel = (TextView)findViewById(R.id.helloworldlabel);

webView.setWebViewClient(new HelloWorldWebViewClient(helloWorldLabel));

And add any dependencies Android Studio tells you that you need to add.

* 1. Note that the HelloWorldWebViewClient portion of that 2nd line is in red. That's because it doesn't exist. We're going to make Android Studio create it for us. Click somewhere in on that portion of the code for that line, and, you'll get a red lightbulb to the left. Click on it and select "Create Class "HelloWorldWebViewClient"" from the context menu. A dialog will pop up. Leave the defaults and click on OK. Android Studio will create the class in the com.softsource.helloworldandroidapp package. You'll get prompted to confirm that you want to use "TextView helloWorldLabel" as a type and parameter name for the constructor. Go ahead and use it.

* 1. Add the @NotNull annotation to the constructor parameter so that it now reads like

@NotNull TextView helloWorldLabel

The NotNull part of @NotNull will be red, indicating that it needs to be implemented as something. Click somewhere on the NotNull part and the red lightbulb will show up again. Click on it and select "Add annotations.jar to classpath". Use the defaults in the "Add Library to Project" dialog that ensues, and click OK. The @NotNull annotation should be happy now and will have turned yellow.

* 1. We've got one more little bit of work to do to get our project to build with the annotation we added. Find the build.gradle file that lives at the same level as the src folder you've been working in, and open it - NOT the one at the topmost level of the project - and open it. Near the bottom of this file, you'll see an empty dependencies definition:

dependencies {

}

Add a nudge to Gradle to build your project with the annotations.jar by adding the following compile line, so that your dependencies setting now looks like this:

dependencies {

compile 'com.intellij:annotations:+@jar'

}

Save that and lets continue.

* 1. Add the following private member to your HelloWorldWebViewClient class, right underneath its declaration again:

private TextView helloWorldTextView;

Then assign the TextView getting passed in at construction time to it, so that now your constructor looks like:

public HelloWorldWebViewClient(@NotNull TextView helloWorldLabel) {

helloWorldTextView = helloWorldLabel;

}

* 1. Now we can work on our intercept. To do so, we need to implement an override of one of the WebViewClient methods. Add the following code directly below the constructor in your HelloWorldWebViewClient class:

@Override

public boolean shouldOverrideUrlLoading(WebView view, String url) {

if(Uri.parse(url).getHost().endsWith("google.com")) {

Log.d("INTERCEPT - Request intercepted:", url);

helloWorldTextView.setVisibility(View.VISIBLE);

return true;

}

return false;

}

And appease the Android Studio dependency cops by approving all dependencies that need to be added. This intercept, as you can see, is a little different. Instead of the whitelisting we did in iOS, we're blacklisting the thing we don't want to allow to load, google.com.

Do a clean and rebuild of your project (under Tools in the Android Studio menu), and run it now. When you click on your button, your newly added label appears!

1. Add Native to Javascript Bridge for Android
   1. This is going to be easy peasy. Just like before, we need to add a button to your MainActivity view and implement it. Within the RelativeLayout element where your TextView lives, add the following block after your Text View:

<Button

android:id="@+id/helloworldbutton"

android:text="@string/change\_title"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_centerHorizontal="true"

android:layout\_alignParentBottom="true"

android:layout\_marginBottom="100dp"

android:onClick="fireJavascriptCommand"/>

Add the following to your strings.xml:

<string name="change\_title">Change the Title</string>

And add the following method to your MainActivity class, just above the PlaceholderFragment internal class in there:

public void fireJavascriptCommand(View view) {

webView.loadUrl("javascript:helloFromNativeCommand()");

}

Fire it up again. You should now see a button just above the HTML button labelled "Change the Title", and clicking on it will execute the same "helloFromNativeCommand" javascript function you wrote before, changing the header.

Success!!!

You've just written a cross platform "Hello World" that can talk back and forth between javascript and native code!