IBM

developerWorks Technical topics Mobile development Technical library

Maqetta means mockup, Part 3: Deploy your Maqetta UI prototype with PhoneGap

Use PhoneGap to deploy a Maqetta UI prototype to any mobile device

So far in this series introducing Maqetta, you've learned how to create and enrich an interactive mobile UI prototype using features from the Dojo and Dojo Mobile toolkits. Working in your browser with Maqetta, you were able to build a believable prototype without writing any code, then augment its features and flow with custom JavaScript. Now it's time to combine Maqetta with PhoneGap in order to create and deploy a new mobile UI prototype onto actual devices.

Tony Erwin is a Software Engineer in IBM's Emerging Internet Technologies group and a core member of the Maqetta development team. Tony has been with IBM since 1998 and has extensive UI design and development experience using a wide variety of technologies and toolkits. Before joining IBM, Tony earned an MS in Computer Science from Indiana University and a BS in Computer Science from Rose-Hulman Institute of Technology.

06 June 2013

Also available in Russian Japanese

Introduction

Combining Maqetta and PhoneGap results in a powerful platform for quickly prototyping, developing, and deploying mobile applications. In this final article in the series, we'll build a new application prototype from scratch, taking a Maqetta mobile application from implementation to deployment. The example application for this article will be a mobile GPS locator app, whose purpose is to allow a user to see his or her current position on a map. First, we'll construct a UI prototype in the Maqetta page editor without writing any code (as we did in Part 1). Then, we'll add some JavaScript that enriches the prototype with more interactive features (as we did in Part 2). We'll then add a new layer to our JavaScript that will enable us to leverage features from PhoneGap's Geolocation API.

About this series

This series shows you how to use Maqetta to prototype HTML5 user interfaces.

In Part 1, learn about Maqetta's major features while creating a prototype for a rich mobile application.

In Part 2, take your prototype application to the next level by writing custom JavaScript to add interactive functionality.

In Part 3, use PhoneGap to turn a Maqetta-generated mobile prototype into a native app that is ready to deploy to actual devices.

Learn more about using Maqetta by reading Tony's blog on developerWorks.

Once the prototype is fully developed, we'll test the location feature with the Ripple Emulator extension for Google Chrome and use the Adobe PhoneGap Build service to turn our Maqetta-produced GPS locator into a native app. We'll conclude by testing the GPS locator's functionality with the Android Emulator, which is part of the Android SDK.

Build on what you know

If you've followed along with this series from the beginning, building and enriching the GPS locator prototype should be a snap — and a good opportunity to cement what you've already learned about Maqetta. If you already have a mobile application in Maqetta that you simply want to build with PhoneGap and deploy to a mobile device, then you can skip ahead to "Export the app to PhoneGap."

Construct the GPS Locator prototype

Our first step is to build the GPS locator UI prototype. The GPS locator will allow users to see their geographic position on a map, switch the style of map, view their current latitude and longitude, and press a button to update to a new position.

Just as we did in Part 1, we'll start by creating a new project in Maqetta and building an HTML file:

1. Log in to the Maqetta server at Maqetta.org. (This step presumes that you have registered as a user on the Maqetta website.)
2. Create a new project by choosing the **Create > Project...** menu option. Enter "gpsLocator" for your project name, and click the **Create** button.
3. Create a new mobile application in the new project by choosing the **Create > Mobile Application...** menu option. Enter "index.html" as your file name (it's the name required by PhoneGap), and click the **Create** button.

Start with the view

The first step to designing the UI is to create the view, which for this application will be implemented using a Dojo Mobile `ScrollableView` widget.

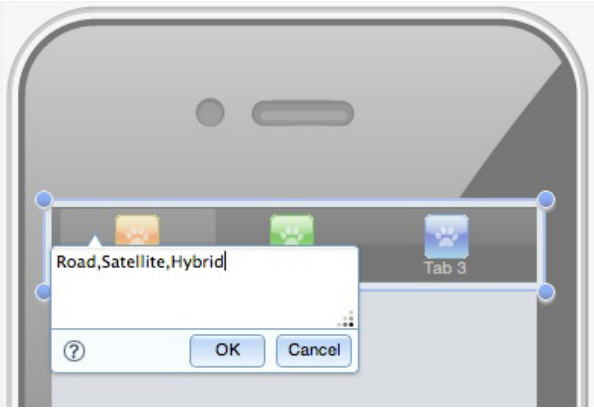
1. Drag and drop a `ScrollableView` widget from the Views section of the widget palette into the middle of the Maqetta device silhouette. (If you have trouble finding a widget, you can type its name into the search box at the top of the widget palette.)
2. In the Properties palette, change the `ID` of the new view to `mainView`.

Add a segmented tab bar

Next, we'll create a segmented tab bar that the user will be able to use to change the type of map that he or she sees.

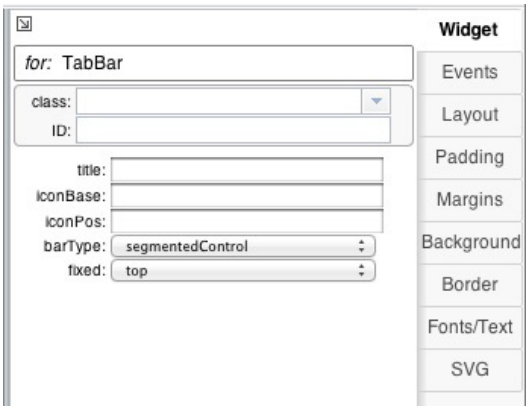
1. Drag and drop a `TabBar` widget from the **Controls > Toolbars, ButtonBars** section of the widget palette on top of the current view.
2. Change the value in the text box to "Road, Satellite, Hybrid" (as shown in Figure 1) and hit **OK**.

Figure 1. Create the TabBar widget



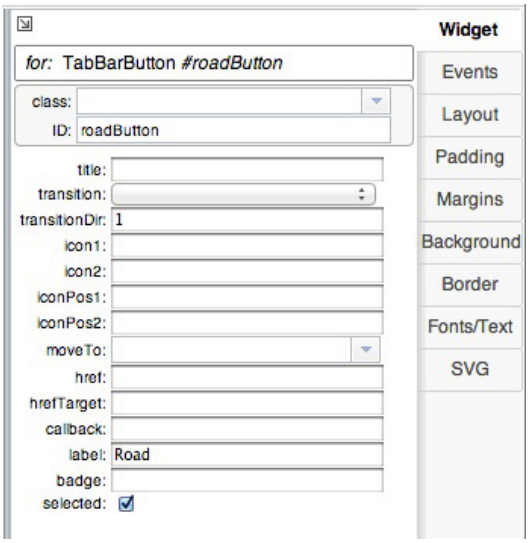
3. In the Widget section of the Properties palette, change `barType` to `segmentedControl` and `fixed` to `top` as shown in Figure 2.

Figure 2. TabBar properties



4. Select the first button in the `TabBar`, which is labeled "Road." In the Properties palette, change the `ID` to `roadButton` and remove the icon strings in the `icon1` and `icon2` attributes (as shown in Figure 3).

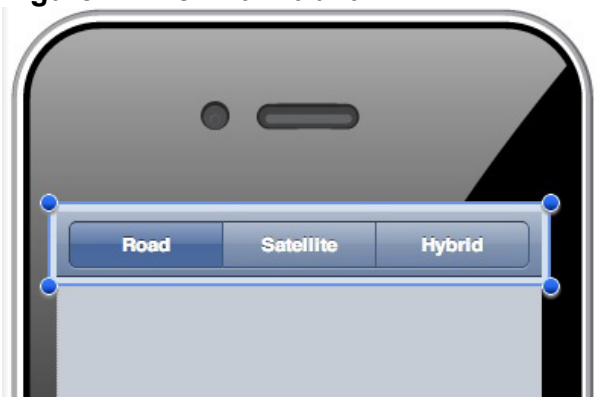
Figure 3. TabBarButton properties



5. Select the second button in the `TabBar`, which is labeled "Satellite." In the Properties palette, change the `ID` to `satelliteButton` and remove the icon strings in the `icon1` and `icon2` attributes.
6. Select the third button in the `TabBar`, labeled "Hybrid." In the Properties palette, change the `ID` to `hybridButton` and remove the icon strings in the `icon1` and `icon2` attributes.

After completing these steps, your tabbed bar should look like the screenshot in Figure 4.

Figure 4. The final TabBar

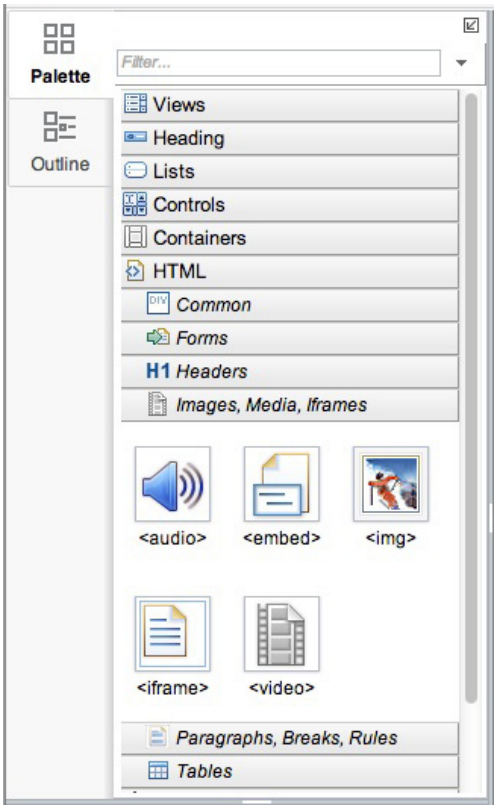


Create the map display

Next, we'll set up a placeholder component for our GPS locator map. The map will ultimately show application users where they are located.

1. Drag and drop a `RoundRect` widget from the **Containers > Dojo** section of the widget palette on top of the current view below the `TabBar`.
2. Open the HTML section of the widget palette, which shows several sections for the many HTML5 elements that Maqetta makes available to designers (see Figure 5).

Figure 5. HTML widgets



3. From the **Images, Media, Iframes** sub-section, drag and drop an `<img>` widget onto the `RoundRect`, making it a child element of the rectangle.
4. In the "Select a source" dialog that comes up, copy and paste the URL from Listing 1 into the File URL field and click **OK**. This URL uses the Google Static Maps API to retrieve a map centered on IBM Corporate Headquarters.

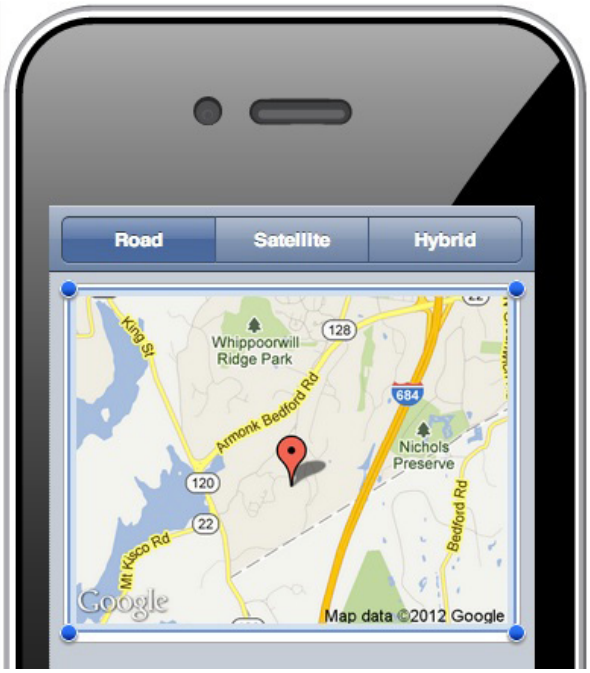
Listing 1. The map URL (copy and paste as a single line)

```
http://maps.googleapis.com/maps/api/staticmap?
markers=41.108556,-73.720729&zoom=13&size=284x216
&sensor=false&scale=2
```

5. In the Layout section of Properties palette, change `width` to 100% and `height` to 44%.
6. Change the `ID` of the `<img>` to `mapImg`.

After completing these steps, your map image should look similar to Figure 6.

Figure 6. Map image

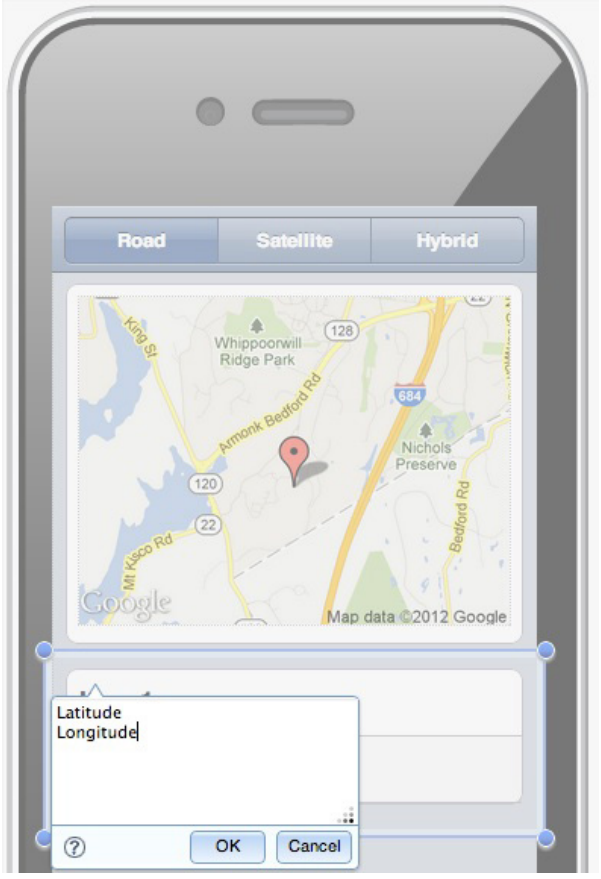


Add latitude and longitude

We'll need to add another widget to display the current latitude and longitude reflected in our map:

1. From the Lists section of the widget palette, drag and drop a `RoundRectList` below the `RoundRect` in your view. Enter "Latitude" on one line and "Longitude" on the next line (as shown in Figure 7), then click **OK**.

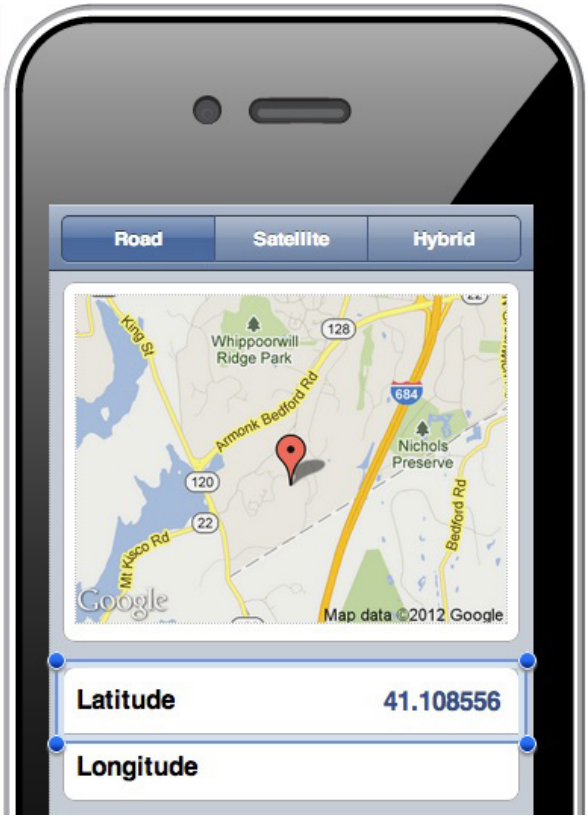
   Figure 7. RoundRectList



2. Select the first list item, labeled "Latitude." In the Widget section of the Properties palette, change the `ID` to `latitudeListItem` and `rightText` to "41.108556" (the latitude value that was in the URL we put

in the map image). After doing so, your list item should look like it does in Figure 8.

Figure 8. Latitude list item



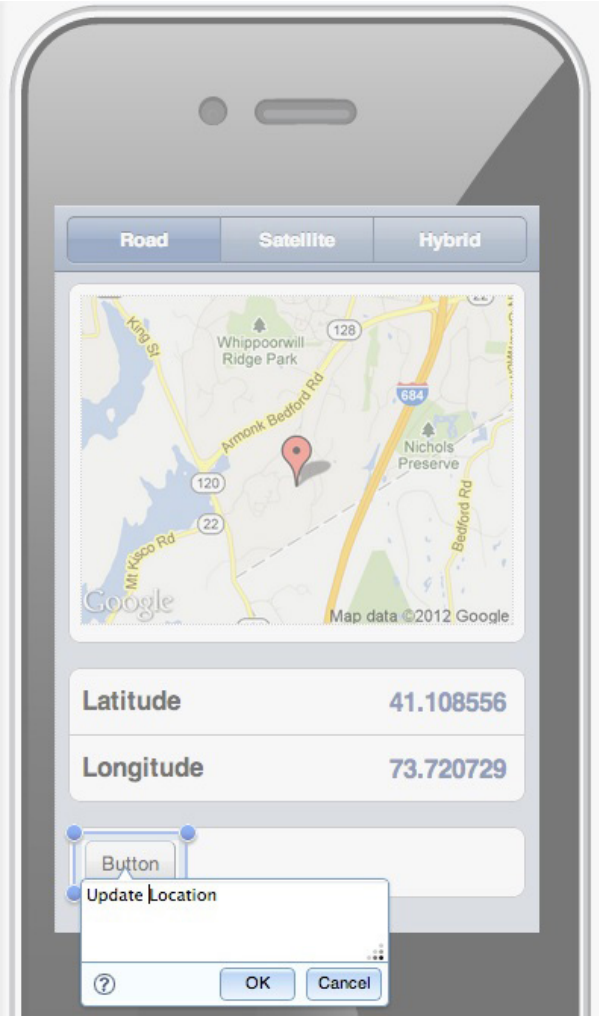
3. Now select the second list item labeled "Longitude." Change the item's `ID` to `longitudeListItem` and its `rightText` to "73.720729" (the longitude in the map image URL).

Location updates

Finally, let's complete the view by adding a button that will allow the user to update his or her location on the map:

1. Drag and drop a `RoundRect` from the **Containers > Dojo** section of the widget palette onto the view underneath the `RoundRectList`.
2. Drag and drop a `Button` from the **Controls > Buttons** section of the widget palette onto the `RoundRect`. Enter "Update Location" into the text box (as shown in Figure 9) and hit **OK**.

Figure 9. Adding the Update Location button

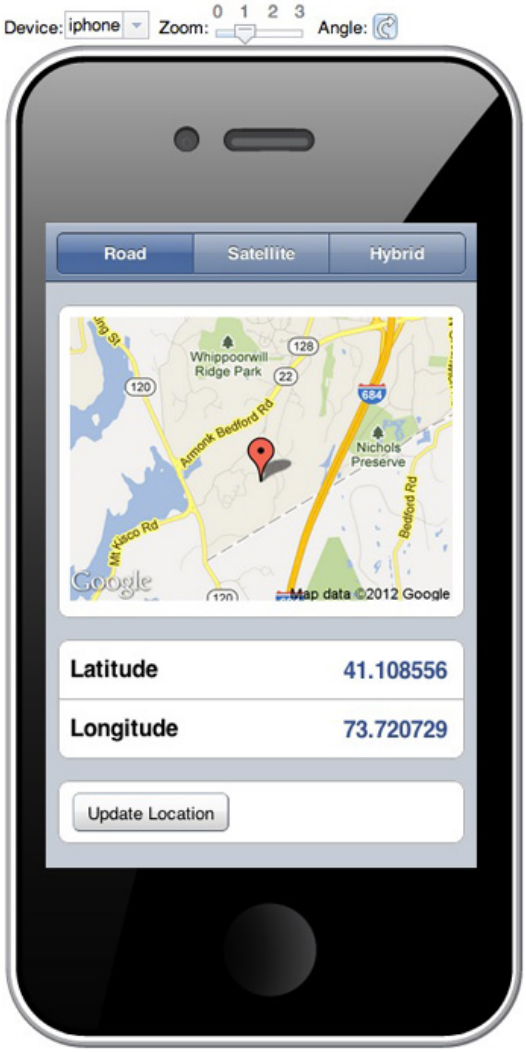


3. In the Properties palette, change the `ID` of the new button to `updateLocationButton`.

Preview the prototype

At this point, we've built a nice mockup without writing any code. You should run the live preview to check it out. Recall from Part 1 and Part 2 that you can run a live preview of any Maqetta prototype by clicking the **Preview in Browser** icon in the Maqetta toolbar. When you do that with the GPS locator, you'll see a new browser tab containing something like the screenshot in Figure 10.

Figure 10. Live preview of the GPS locator app



Assuming that your prototype looks as good as mine, what we have is sufficient, at least for getting feedback from potential users. Note the shortcomings in the UI interaction, however:

- The initial position (location on a map) is hard-coded and has nothing to do with the user's actual position.
- Clicking on the **Road**, **Satellite**, or **Hybrid** buttons does not change the map type.
- Clicking on the **Update Location** button does not change the latitude or longitude list items; coordinates shown on the map remain the same.
- Changing the orientation of the device silhouette results in a distorted map. This is because the URL sent to Google has a hard-coded size, and the `<img>` element dimensions change according to device orientation.

Just as we did in Part 2 of this series, we'll add some custom JavaScript to correct these limitations and improve the interactivity of the GPS locator prototype.

Enrich the app with JavaScript

Hopefully you've already had some experience (perhaps in Part 2) with working in a Maqetta prototype's `app.js` file, adding custom JavaScript to enhance a UI's interactivity. In that case, the steps in this section will confirm much of what you already know. Follow along by copying and pasting the individual

JavaScript code snippets here into your own `app.js` file. Many of the code snippets represent testable enhancements to the app, so I'll prompt you to preview the application changes as we go along. If you prefer a faster track, you can just download the final version of my `app.js` and replace yours with it in the Maqetta workspace.

Required modules

As in Part 2, we'll start by specifying the Dojo modules that we're going to need. To accomplish this, replace your `app.js` with the code in Listing 2. (Recall that you can open `app.js` by double-clicking it in the Files palette.)

Listing 2. Specifying Dojo modules

```
/*
 * This file is provided for custom JavaScript logic that your HTML files might need.
 * Maqetta includes this JavaScript file by default within HTML pages authored in
 * Maqetta.
 */
require(["dojo/ready",
        "dojo/dom",
        "dojo/dom-style",
        "dijit/registry",
        "dojo/on",
        "dojo/aspect"],
function(ready, dom, domStyle, registry, on, aspect){
     ready(function(){
        // logic that requires that Dojo is fully initialized should go here

     });
});
```

The above code loads the following modules:

- dojo/ready
- dojo/dom
- dojo/dom-style
- dijit/registry
- dojo/on
- dojo/aspect

Referencing widgets

Next we need to get references to all of the widgets we're going to interact with in our code. As shown in Listing 3, we'll use `registry.byId` to reference Dojo widgets and `dom.byId` for the DOM element —, which is the <img> element holding our map. We'll also insert an `if` block to ensure that all of the widgets were found. If not all widgets were found, we'll get an alert message that will make it easier to discover and correct the problem.

Listing 3. Widget references

```
        /* ****************************************
         * Get a reference to all the widgets and DOM
         * elements we need.
         ****************************************/
        var mainView = registry.byId("mainView");
        var roadButton = registry.byId("roadButton");
        var satelliteButton = registry.byId("satelliteButton");
        var hybridButton = registry.byId("hybridButton");
        var updateLocationButton = registry.byId("updateLocationButton");
        var latitudeListItem = registry.byId("latitudeListItem");
        var longitudeListItem = registry.byId("longitudeListItem");
        var mapImg = dom.byId("mapImg");

        // Make sure we found all of the widgets
        if (!mainView ||
            !roadButton ||
            !satelliteButton ||
            !hybridButton ||
```

```
            !updateLocationButton ||
            !latitudeListItem ||
            !longitudeListItem ||
            !mapImg) {

            // Trace out an error to make it easier to figure out
            // which widget(s) could not be found
            alert("could not find at least one of the widgets:\n" +
                "\t mainView = " +  mainView + ",\n" +
                "\t roadButton = " +  roadButton + ",\n" +
                "\t satelliteButton = " +  satelliteButton + ",\n" +
                "\t hybridButton = " +  hybridButton + ",\n" +
                "\t updateLocationButton = " +  updateLocationButton + ",\n" +
                "\t latitudeListItem = " +  latitudeListItem + ",\n" +
                "\t longitudeListItem = " +  longitudeListItem + ",\n" +
                "\t mapImg = " +  mapImg);

            // return, so don't run any other JavaScript
            return;
        }
```

You can copy and paste this code immediately inside of the `ready` function in your `app.js` file. You can then save the file and preview the app to make sure that all of the widgets were found.

Add a map data placeholder

Next, in Listing 4, we define a variable called `mapData`. This will act as a placeholder for the map data that is required to build a URL, which will be used to retrieve a map. We'll initially populate this placeholder with default data, but as the user interacts with the app, we'll change its data and use it to populate the widgets on the page. You may notice that `latitude` and `longitude` are set to `undefined` because we don't yet know the user's initial position. Go ahead and add this code to your `app.js` file; just note that the prototype's behavior won't change in the preview.

Listing 4. Map data

```
        /* ****************************************************
         * Initialize our placeholder for all of the map data
         * we want to pass to Google.
         ******************************************************/
        var mapData = {
            latitude: undefined,
            longitude: undefined,
            zoom: 13,
            width: 284,
            height: 216,
            sensor: false,
            scale: 2,
            mapType: "roadmap"
        }
```

Build the map URL

Next we'll define a function called `getMapUrl` that builds a URL using the Google Static Maps API (see Listing 5). The `getMapUrl` function uses `mapData` to fill in the required parameters for a given URL. Before building the URL, it checks to see if the user's latitude and longitude have been set. If not, it will default the location coordinates to 0, 0 (with a zoom level of 1), so we'll build a URL to retrieve a zoomed-out world map. Seen on a display, this setting will give the user an indication that we don't know his or her actual position yet.

Listing 5. Get map URL

```
        /* ****************************************************
         * Function to build the Google url for the map we
         * want to retrieve. It fills in the url parameters
         * based on data in our "mapData" variable.
         ******************************************************/
        var getMapUrl = function() {
            // If we don't have a valid latitude/longitude, set up
            // parms to get a world map centered at 0, 0 with zoom
            // level of 1.
            var latitude = mapData.latitude ? mapData.latitude : 0;
            var longitude = mapData.longitude ? mapData.longitude : 0;
            var zoom = mapData.latitude ? mapData.zoom : 1;

            // Build the url
            var url =
                "http://maps.googleapis.com/maps/api/staticmap?" +
                "markers=" + latitude + "," + longitude + "&" +
                "zoom=" + zoom + "&" +
                "size=" + mapData.width + "x" + mapData.height + "&" +
```

```
            "sensor=" + mapData.sensor + "&" +
            "scale=" + mapData.scale + "&" +
            "maptype=" + mapData.mapType;

        return url;
    };
```

We're not calling this function yet, so when you add it to `app.js`, there will be no change in the behavior of the prototype.

Update the widgets

Next, we'll put in place a function called `updateWidgets` (see Listing 6). When it's called, it refreshes the contents of the widgets on the page. It first uses `getMapUrl` and sets the `src` of the `mapImg` with the result. It then updates `latitudeListItem` and `longitudeListItem` with the latitude and longitude values in `mapData`. If the latitude and longitude haven't been set, a placeholder ("--") will be displayed in each of the list items.

Listing 6. Function for updating widgets

```
    /* ****************************************************
     * Function to updateWidgets on the page based on the
     * current mapData.
     ******************************************************/
    var updateWidgets = function() {
        //Get the map URL and update the image
        var url = getMapUrl();
        mapImg.src = url;

        //Update the latitude on the display
        var latitudeString =
                mapData.latitude ? mapData.latitude.toFixed(6) : "--";
        latitudeListItem.set("rightText", latitudeString);

        //Update the longitude on the display
        var longitudeString =
                mapData.longitude ? mapData.longitude.toFixed(6) : "--";
        longitudeListItem.set("rightText", longitudeString);
    };
```

Update `app.js` with this code snippet, but again there will be no change in the behavior of the preview.

Button handlers for map type

One of the current limitations to the prototype is that clicking the **Road**, **Satellite**, or **Hybrid** buttons has no effect. Our previous changes to the prototype's JavaScript enable us to fix this issue in a straightforward manner. In Listing 7, we register a function to be called when any one of the three buttons is clicked. In each handler, we then set the `mapType` in `mapData` to the appropriate value. Finally, we call the `updateWidgets` function that was described in the previous section.

Listing 7. Button handlers

```
    /* ****************************************************
     * Handle buttons in segmented tab control
     ******************************************************/
    // Change the mapType and update widgets when roadButton is clicked
    on(roadButton, "click", function() {
        mapData.mapType = "roadmap";
        updateWidgets();
    });

    // Change the mapType and update widgets when satelliteButton is clicked
    on(satelliteButton, "click", function() {
        mapData.mapType = "satellite";
        updateWidgets();
    });

    // Change the mapType and update widgets when hybridButton is clicked
    on(hybridButton, "click", function() {
        mapData.mapType = "hybrid";
        updateWidgets();
    });
```

When you update `app.js` with the code in Listing 7, you should see a clear change in the behavior of the prototype. The initial display will still contain the map of IBM Corporate Headquarters (we'll fix this shortly). But, when you click on any of the three buttons, a new map URL will be built using the `mapType` for the

clicked button. Since we haven't set the user's latitude and longitude, the URL will automatically retrieve the zoomed-out world view. Figure 11, for example, shows the world map that results from clicking the **Satellite** button. Note also that the **latitude** and **longitude** list items have been filled in with a placeholder ("--").

Figure 11. Changing the map type

Changing image dimensions

Another limitation to the original prototype is the distortion in the map image when the dimensions of `mapImg` are changed (such as when device type or orientation is changed). This happens because the URL used to retrieve the map requires the image height and width in pixels. We obviously don't know those values for all devices and orientations, so how should we fix this?

Listing 8 demonstrates a solution that involves using `aspect.after` to cause a function to run after `mainView`'s `resize` function has been called. In the function specified, we use `domStyle.get` to get the calculated width and height of `mapImg` and put the updated values back into `mapData`. We then call `updateWidgets`, which ultimately causes a new URL to be built and set on `mapImg`.

Listing 8. Handling resize events

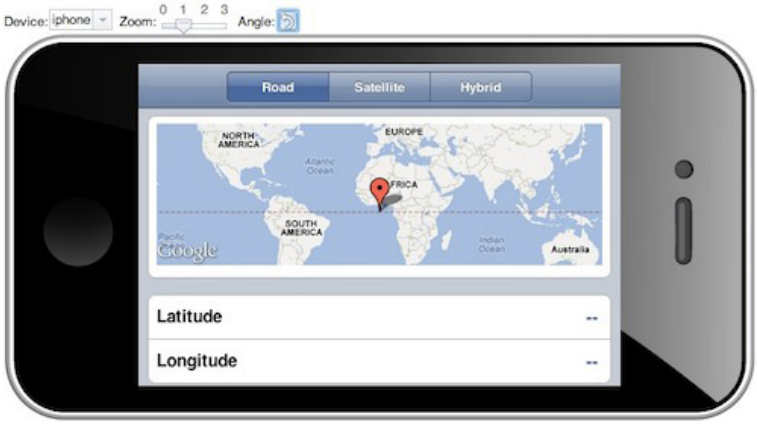
```
        /* ****************************************************
         * Let's monitor changes in the dimensions of our
         * mainView (such that might occur during a change in
         * orientation). This will allow us to get the new
         * dimensions of our mapImg and allow us to make a
         * new request for the right sized map.
         ****************************************************/
        aspect.after(mainView, "resize", function() {
```

```
            // Update map data based on actual width/height of mapImg
            mapData.width = Math.round(domStyle.get(mapImg, "width"));
            mapData.height = Math.round(domStyle.get(mapImg, "height"));

            // mapData has been changed, so update the widgets
            updateWidgets();
        });
```

Now update `app.js` and preview the app. Because an initial resize occurs up front, `updateWidgets` will be called right away and we'll see the world map immediately. If you change the orientation of the device, the map image will reload with the right size, thus eliminating distortion. Note, however, that after the orientation changes you will now need to scroll to see the **Update** button. You could fix this by refining the resize handler to change the height of `mapImg`, so that everything would fit without scrolling.

Figure 12. A more seamless orientation change



With that, we've updated our prototype, using custom JavaScript to make some of its key features more responsive to user input. So far everything we've done reflects what you probably already learned in Part 1 and Part 2 of this series. In the next section, we'll start incorporating new functionality that will enable us to work with PhoneGap APIs.

PhoneGap and Maqetta

An important feature of the GPS locator is the ability to update the location shown on a map based on the user's physical location. We also ultimately want to turn this prototype into a native mobile application built with PhoneGap. When the app is running natively on a device, we'll have access to all of the PhoneGap APIs. These APIs make it possible to access native services of mobile devices (such as location, accelerometer, and camera) via a common JavaScript layer. We'll be specifically using PhoneGap's Geolocation API, which conforms to the W3C Geolocation API specification. The desktop browsers supported by Maqetta — Chrome, Firefox, and Safari — all implement this spec. As a result, when we write our code, we'll be able to access the user's physical location in the Maqetta previewer, even though PhoneGap will not be present.

With that background in mind, let's turn our attention to the `updateCurrentPosition` function in Listing 9. The first thing we'll do is disable the **Update Location** button. We do this because it can sometimes take a bit of time to acquire a device's GPS position, and we don't really want the user clicking the button multiple times. Then, we invoke `navigator.geolocation.getCurrentPosition` from the Geolocation API. This function works asynchronously, so we need to pass in a function that will get called when the location is determined. The function we pass in is `onGetPositionSuccess` (we also pass in `onGetPositionError`, which will be called if an error occurs).

When `onGetPositionSuccess` is called, it receives a `Position` object with all sorts of information about the user's location — aside from latitude and longitude, it also contains information about altitude, heading, speed, and more. We simply update the `latitude` and `longitude` variables in `mapData` and then call the `updateWidgets` function to refresh the display. We also re-enable the **Update Location** button, because it's now safe for the user to make another request.

We don't have any code (yet) that invokes `updateCurrentPosition`. So, after updating `app.js` with the code below, there will be no change to the app's preview behavior.

Listing 9. Getting the map position

```
/* ***************************************************
 * Callback function when GPS successfully acquired from
 * PhoneGap API (specified in "updateCurrentPosition"
 * function below). It accepts a `Position` object, which
 * contains the current GPS coordinates/
 ***************************************************/
var onGetPositionSuccess = function(position) {
    mapData.latitude = position.coords.latitude;
    mapData.longitude = position.coords.longitude;

    // mapData has been changed, so update the widgets
    updateWidgets();

    // Re-enable updateLocationButton
    updateLocationButton.set("disabled", false);
};

/* ***************************************************
 * Callback function when GPS acquisition fails using
 * PhoneGap API (specified in "updateCurrentPosition"
 * function below). It accepts a PositionError object
 * containing information on the failure.
 ***************************************************/
var onGetPositionError = function(error) {
    //Show an alert with the error
    alert('geolocation failure! code: ' + error.code    + '\n' +
           'message: ' + error.message + '\n');

    // Re-enable updateLocationButton
    updateLocationButton.set("disabled", false);
}

/* ***************************************************
 * Function used to get the new GPS position based on
 * the environment we're running in (PhoneGap or not).
 * Widgets will be updated one the GPS position is
 * acquired.
 ***************************************************/
var updateCurrentPosition = function() {
    if(navigator.geolocation && navigator.geolocation.getCurrentPosition){
        // Disable update location button while we're getting location
        updateLocationButton.set("disabled", true);

        // Geolocation services are available so request the
        // position. The geolocation API is asynchronous, so
        // we have to rely on a callback function.
        navigator.geolocation.getCurrentPosition(
            onGetPositionSuccess, //function to call when GPS location acquired
            onGetPositionError, //function to call if location cannot be acquired
            { enableHighAccuracy: true }); //added for Android simulator quirk
    } else {
        // We don't have geolocation services, so show an alert
        alert("Geolocation services are not available!");
    }
};
```

Update Location button handler

We're now set up to rather easily respond to clicks of the **Update Location** button. In Listing 10, we register a handler that simply calls the `updateCurrentPosition` function from the previous section.

Listing 10. Handling the Update Location button

```
/* ***************************************************
 * Initiate update of the current position when
 * updateLocationButton is clicked
 ***************************************************/
on(updateLocationButton, "click", function() {
    updateCurrentPosition();
});
```

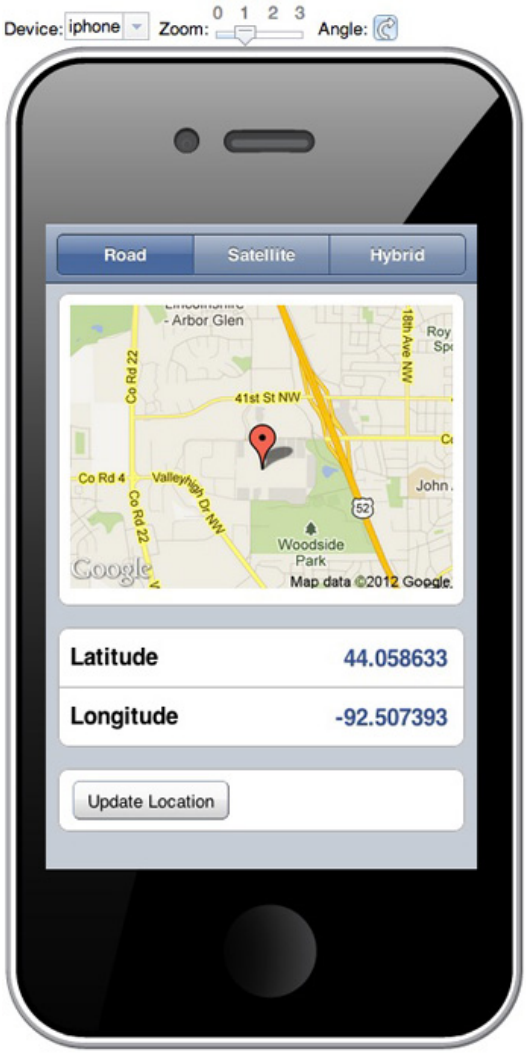
At this point, you can update `app.js` and preview the app. Now, if you click the **Update Location** button,

the button will gray out and your browser will ask you if you want to grant permission for the application to have access to your location.

Figure 13. Update Location browser prompt

Then, if you choose to the share your location, the asynchronous call to the `onGetPositionSuccess` callback function will be made, causing an update to the information on the display with your retrieved location (see Figure 14). In addition, the **Update Location** button is re-enabled.

Figure 14. Updated Location in Maqetta Preview



Device initialization

We've implemented the bulk of the prototype's functionality. The one major thing we're still missing is the ability to initialize the user's position without requiring him or her to press the **Update Location** button. In order to get the user's current position, we need to be able to call PhoneGap APIs. But before we can safely call PhoneGap APIs we need to know when the device is "ready." While we want to run our app natively on mobile devices, we'd also like to be able to continue to run in the Maqetta previewer for easier design and prototyping.

As you saw in the Maqetta previewer, the browser environment is ready by default, and we can immediately call the Geolocation API. When running in a PhoneGap environment, however, we need to listen for a special event that signals device readiness. So, for the remainder of this section, we'll focus on writing some JavaScript that will allow us to do the following:

1. Check whether PhoneGap is present
2. Use that result to check whether the device is ready
3. Once the device is ready, initialize the user's position

Device readiness and initialization

First, let's define two functions (in Listing 11) that we'll use to facilitate initialization. The first, `beforeDeviceReady`, sets up the widgets and disables the **Update Location** button. We'll call this as a

first step *before* the device is ready. We'll call the second function, `onDeviceReady`, immediately *after* we determine that the device is ready. The `onDeviceReady` function will then call `updateCurrentPosition` to automatically get the location on the user's behalf. Add these functions to your `app.js` now, but note that there will be no change in the preview.

Listing 11. Initialization functions

```
/* ****************************************************
 * Function to do some initialization prior to the device
 * being ready.
 ******************************************************/
var beforeDeviceReady = function() {
    // Update widgets with the default values
    updateWidgets();

    // updateLocationButton disabled until device ready
    updateLocationButton.set("disabled", true);
}

/* ****************************************************
 * Function to be called when the device is ready and
 * we can safely try to get location, etc.
 ******************************************************/
var onDeviceReady = function() {
    // Get the initial position
    updateCurrentPosition();
}
```

Next, we'll define the `isPhoneGap` function, which will help us test whether or not we're in a PhoneGap environment (see Listing 12). Knowing which environment we're in will help us decide how best to determine readiness. Unfortunately, there's not a general consensus about the best way to run an environment check, and the most foolproof approaches are rather complicated. The implementation below should work well enough for our prototype, however. Add the following function to `app.js`, but note that there will be no change in the preview until we add the next snippet.

Listing 12. Checking for PhoneGap

```
/* ****************************************************
 * Function used to determine if we're in a PhoneGap
 * environment or not. Unfortunately, there's not
 * not a consensus on the best way for doing this
 * (e.g., see this discussion at StackOverflow:
 * http://bit.ly/Wvmmcw ). And, our situation is
 * is complicated by our desire to run
 * in Ripple on the desktop. So, I think this code
 * is a reasonable compromise for a prototype.
 *
 ******************************************************/
var isPhoneGap = function() {
    var result =
      // Check for PhoneGap/Cordova works in Ripple, but not
      // reliable until deviceready
      window.PhoneGap ||
      window.Cordova ||
      // check for browser is more reliable, but will mean if go to
      // preview URL from mobile browser, this test will pass and it
      // will think you want to use PhoneGap.
      navigator.userAgent.match(/(iPhone|iPod|iPad|Android|BlackBerry)/);

      return result;
}
```

In Listing 13, we put the functions from the last couple of listings to use. First, we call `beforeDeviceReady` as an initialization step. Then, if we're using PhoneGap, we register a listener for PhoneGap's `deviceready` event. When that event fires, it's an indication that it's safe to start calling PhoneGap APIs, and the `onDeviceReady` function will be called. If we're not in PhoneGap (for instance, if we're in the Maqetta previewer), the device is "ready" by default, so we can immediately call `onDeviceReady`.

Listing 13. Initializing the page

```
/* ****************************************************
 * Bootstrapping -- call beforeDeviceReady and then
 * cause onDeviceReady to be called depending on the
 * environment we're in.
 ******************************************************/
beforeDeviceReady();
```

```
        if (isPhoneGap()) {
            // We're running on with PhoneGap, so register for deviceready.
            document.addEventListener("deviceready", onDeviceReady, false);
        } else {
            // We're in a desktop browser, so our device is automatically "ready"
            onDeviceReady();
        }
```

After you update `app.js` with the code in Listing 13, the preview app should show your current position. You might still be wondering how we'll test the PhoneGap branch and **Update Location** functionality, however. In the next section, the Ripple Emulator will answer both of these questions.

The Ripple Emulator

The Ripple Emulator (in beta at the time of this writing) is a Google Chrome extension that provides a browser environment where a subset of the PhoneGap APIs is emulated. We'll use the Ripple Emulator to test calls to the Geolocation API, as well as the app's use of the `deviceready` event.

Set up the Ripple Emulator as follows:

1. Using Google Chrome, go to http://emulate.phonegap.com/.
2. If you've never used Ripple before, you will be advised to install the Ripple Emulator (see Figure 14). Click the **Get Ripple Emulator** button to get started.

   Figure 15. Install Ripple Emulator

3. Next you'll be taken to the Chrome Web Store where you should click the **Add to Chrome** button.
4. Reload http://emulate.phonegap.com/ and you'll see a form to enter a URL to be emulated, as shown in Figure 16.

   Figure 16. Ripple installed (enter a URL)

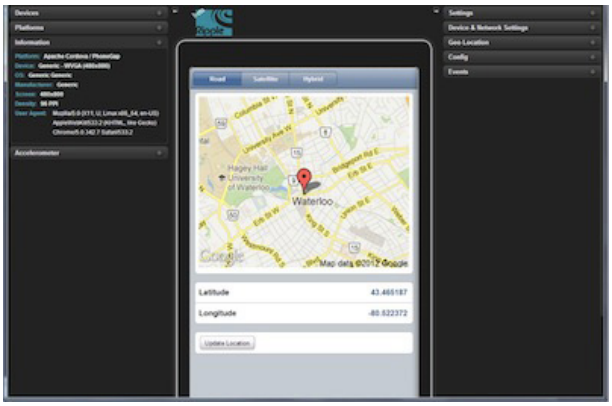
Test the GPS locator

With the Ripple Emulator launched, we're ready to test the app:

1. Enter the URL for the GPS locator and hit **Return**. To get the URL, copy and paste the link next to the **Previewing** label in the Maqetta preview. It should be something like the following but will vary depending on your personal identifier: http://app.maqetta.org/maqetta/user/XYZ/ws/workspace/gpsLocator/index.html.
2. Once Ripple launches, you should see the GPS locator app running in a simulated mobile device, and the map centered on Waterloo, Ontario, Canada (see Figure 17). Since Waterloo is the default position

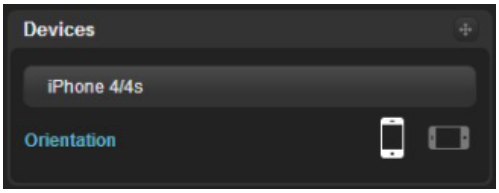
provided by the Ripple Emulator, this indicates that `deviceready` was fired successfully and that our code in `updateCurrentPosition` successfully invoked PhoneGap's `navigator.geolocation.getCurrentPosition` function.

Figure 17. GPS locator's first run in Ripple



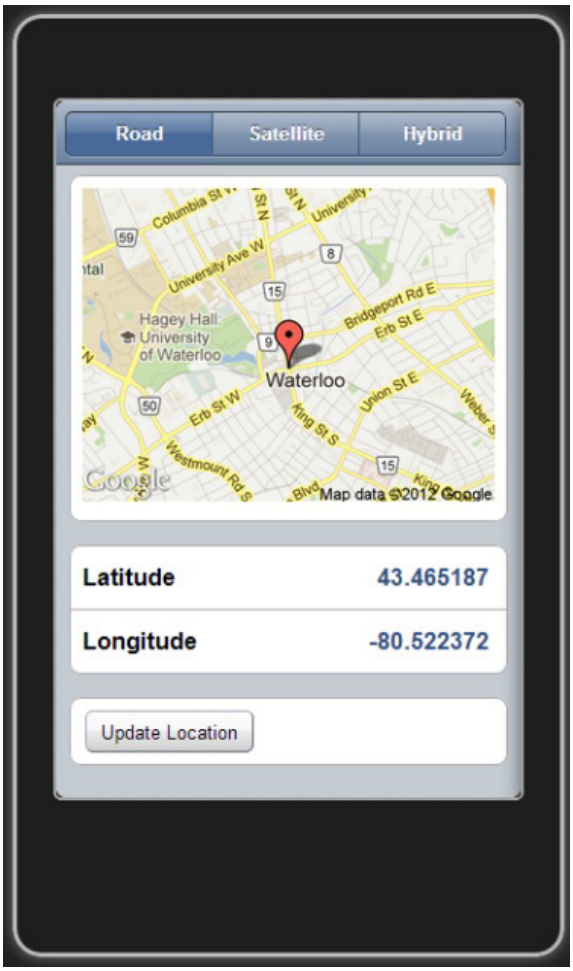
3. To change the device used for the preview, open the Devices section in the upper-left corner of Ripple. The default is set to **Generic – WVGA (480x800)**. Change the selection to **iPhone 4/4S** as shown in Figure 18.

   Figure 18. Change the device setting in Ripple

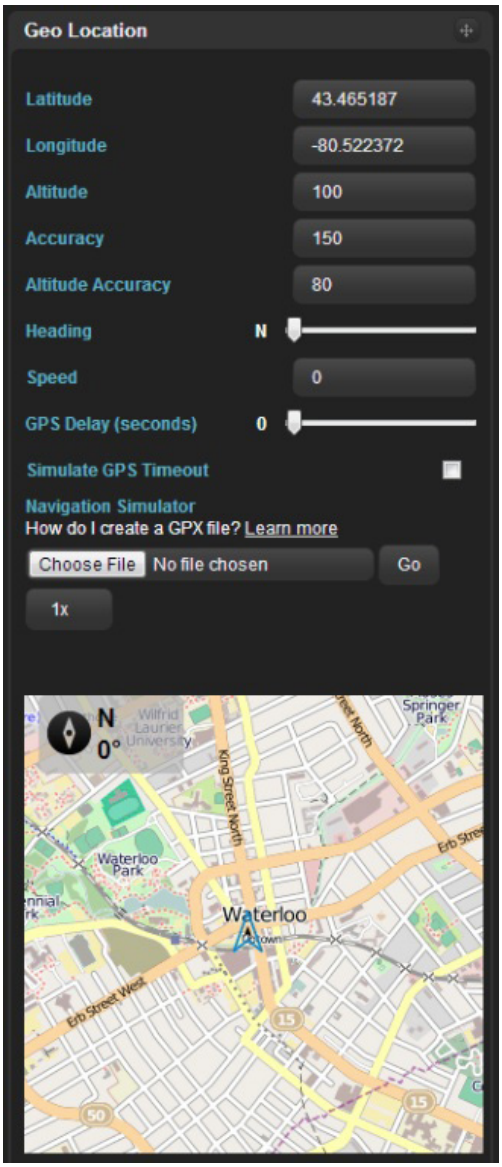


4. Now the size of the device rendering in Ripple should look exactly like it did in the Maqetta preview (see Figure 19).

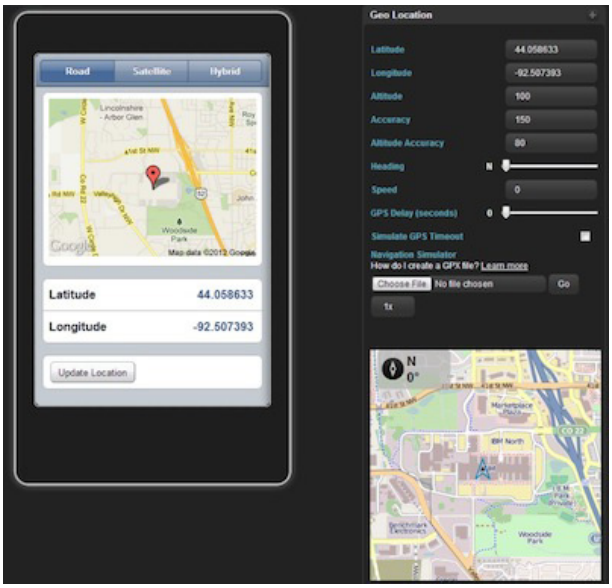
   Figure 19. GPS locator rendered for iPhone in Ripple



5. Open the Geo Location section on the right side of Ripple. As shown in Figure 20, you can see detailed information about the device location being provided to the PhoneGap APIs. You'll notice that the map in the prototype is centered in the same position as the map in Ripple's Geo Location tab. The values that the app is reporting for latitude and longitude also match.
   Figure 20. Location tab in Ripple



6. Experiment with the settings in the Geo Location tab. You can slide the map to a new position or enter explicit values for latitude and longitude. After making your update, click the app's **Update Location** button to see what happens. In Figure 21, the position was updated to the IBM site in Rochester, MN (latitude = 44.058633 and longitude = -92.507393).
   Figure 21. Updated location from Ripple Emulator



Export the app to PhoneGap

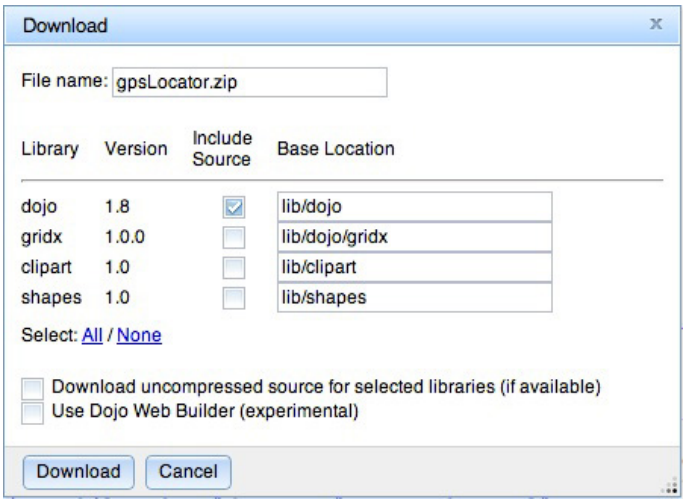
Now, that we've seen that the app is working (including its use of PhoneGap APIs), we can prepare to build it with the Adobe PhoneGap Build service. We'll start by exporting the project from Maqetta, but before doing so, check the following (note that these steps are provided as a checklist for exporting future Maqetta projects to PhoneGap).

1. Ensure that your main HTML file is named `index.html` and is located in the root directory of your project. This file name is required for any Maqetta project that you want to build with PhoneGap.
2. If you are using PhoneGap APIs (as we did for the GPS locator), you'll need to load `phonegap.js` from your `index.html`. This will pull in the JavaScript that the PhoneGap APIs require. First, view the HTML source (recall the **Source** button in the Maqetta toolbar), then add the line in Listing 14 just *above* the line that loads `app.js` (which should be around line 26). Then save your HTML file.
   Listing 14. Loading phonegap.js from index.html
   ```
   <script stype="text/javascript" src="phonegap.js"></script>
   ```
3. Note that even though we're loading `phonegap.js`, the PhoneGap JavaScript resources are not in your Maqetta project (nor do they need to be). They will, however, be included in the set of resources packaged with your native app after building with PhoneGap.
4. Save `index.html` and go back to Design mode by clicking the **Design** button in the Maqetta toolbar.

Export the project zip from Maqetta

Next we'll download the project from Maqetta as a zip file, which we'll then provide to the PhoneGap build service:

1. In the Maqetta Files palette, click the **Download Entire Workspace** icon in the toolbar, which will invoke the Download dialog (see Figure 22). The zip file name is defaulted based on your project name.
2. Uncheck the **gridx**, **clipart**, and **shapes** libraries because we're not using them. Doing this will decrease the size of the zip file.
   Figure 22. Download the project from Maqetta



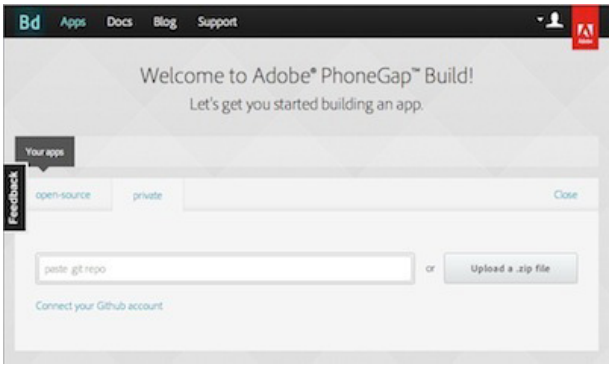
3. Click the **Download** button, which will download the zip file containing the project's resources to your file system.

Build the prototype with PhoneGap

We're now ready to start the process of building the prototype as a native app by using the Adobe PhoneGap Build service. Start by going to https://build.phonegap.com/. Assuming that you've never signed in to PhoneGap before:

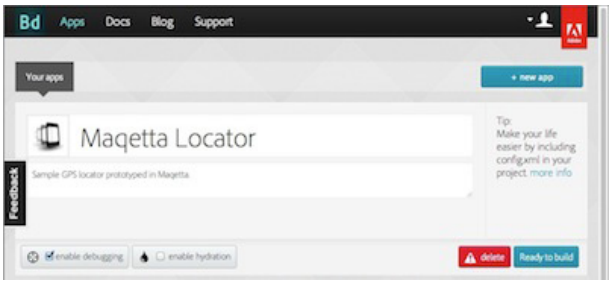
1. Click the **Get started!** button.
2. If you just want to try it out without paying, click **Completely Free**, which allows you to build/manage one private app at a time.
3. Log in with either your Adobe ID or your GitHub credentials.
4. Once you are logged in, you will see a screen like the one in Figure 23 below. Make sure the Private tab is selected, and click the **Upload a zip file** button.

Figure 23. PhoneGap build



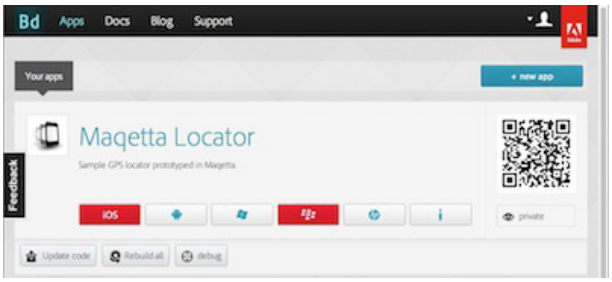
5. Using the file chooser dialog that comes up, select the zip file that you downloaded from Maqetta (e.g., `gpsLocator.zip`).
6. After the file has uploaded you'll see a page like the one in Figure 24, which invites you to enter some information about your app and build it. Enter a name for your app like "Maqetta Locator." You can also enter a description such as "Sample GPS locator prototyped in Maqetta." Given that we're clearly in prototyping mode, it's probably a good idea to check the **Enable debugging** option (using the debugger is beyond the scope of this article).

Figure 24. App panel in PhoneGap



7. Click the **Ready to build** button, and the Adobe PhoneGap Build service will start the process of building native apps for a variety of platforms.
8. Once the build completes, the panel for your app will have a series of buttons (one for each device) to allow you to download the results (see Figure 25). Buttons that are colored red indicate that the build for that particular device did not complete successfully. You can click the red buttons to get more information. Note that both iOS (because we did not provide a developer key) and Blackberry (because there are too many files) will fail. Updating the application for these devices is beyond the scope of this article.

   Figure 25. Build complete



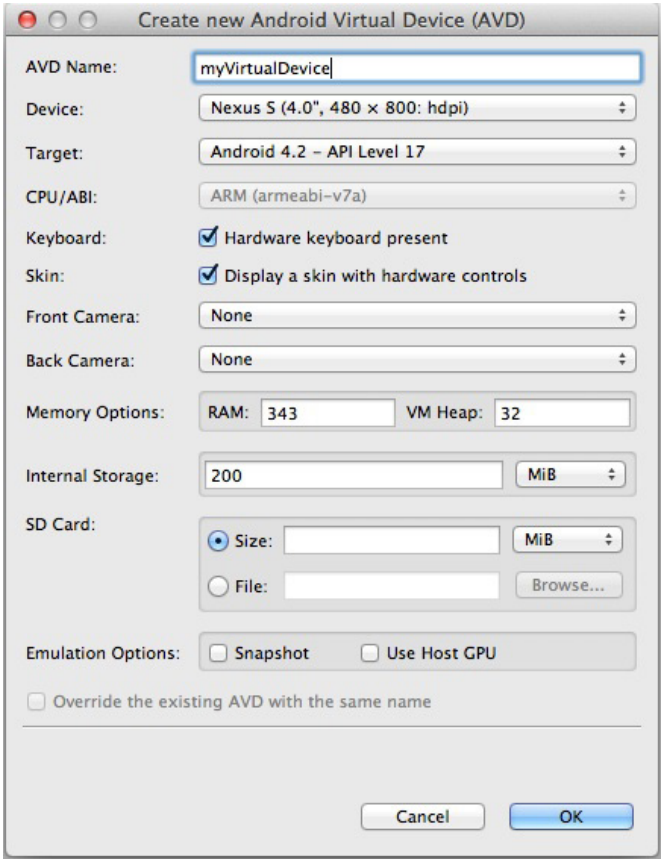
9. If you ever change your app in Maqetta, you can download a new zip file and then click the **Update code** button in PhoneGap to run a new build.

The Android Emulator

One way to test the PhoneGap output without having to access a physical device is to use the Android Emulator, which is part of the Android SDK. If you want to do this yourself, you can follow the steps below to create an Android Virtual Device on which to install your app:

1. Follow the instructions to download and install the Android Developer Tools (ADT) Bundle on your system. It includes the Android SDK, the Android Platform-tools, and a version of the Eclipse IDE with the ADT plugin.
2. After installing the ADT bundle, launch the Eclipse IDE that comes with the bundle.
3. From Eclipse, choose the **Window > Android Virtual Device Manager** menu option.
4. In the dialog that comes up, click the **New...** button to create a new device.
5. In the Create dialog (see Figure 26) give your device a name (such as "myVirtualDevice"), choose a device (I chose "Nexus S"), set the target platform to Android 4.2, and click **OK**.

   Figure 26. Creating a virtual device



6. The dialog will disappear, and the new device will appear in the table of Android virtual devices. Select it and click the **Start...** button. Then click **Launch** in the next dialog. This will start the Android Emulator.

Deploy the app

With the Android Emulator running, we can deploy and test the app:

1. On the PhoneGap build page, click the button to download the app for Android. This will save an .apk file to your file system with a name like `MaqettaLocator-debug.apk`.
2. Once you have the .apk file, you can intall it on your virtual device by using the Android Debug Bridge (`adb`) command from the command line. Your command would look something like the following:

```
<sdk>/platform-tools/adb install <path_to_apk>/MaqettaLocator-debug.apk
```

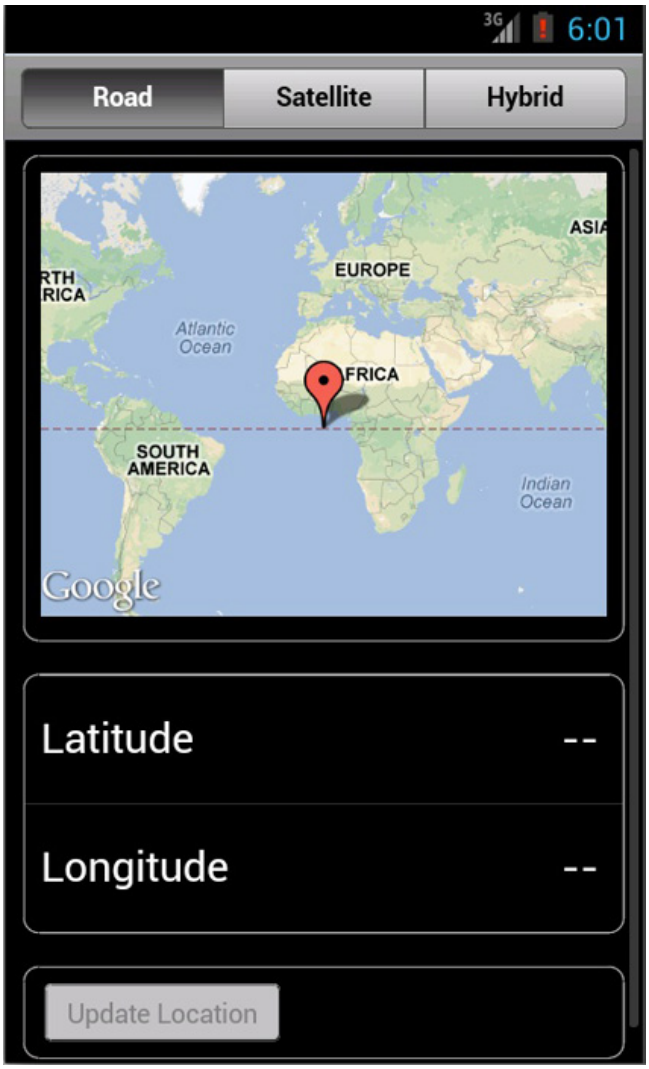
   If this command is successful, you'll get a "Success" message in the console and then be able to see the app represented as an icon alongside all of the other apps installed on the emulator (note the "Maqetta Locator" app icon in Figure 27).
3. To launch the app, just double-click it.

Figure 27. App displayed on PhoneGap's Android Emulator launch screen

Set the location in Android Emulator

As shown in Figure 28, when you launch the app in Android Emulator, you'll see the zoomed-out world map; this is because no geo-position is initially available within the emulator.

Figure 28. GPS locator in Android Emulator before position set



To change this, we need to set the geographic location known to the Android Emulator:

1. From the command line, connect to the Emulator Console with the following command:

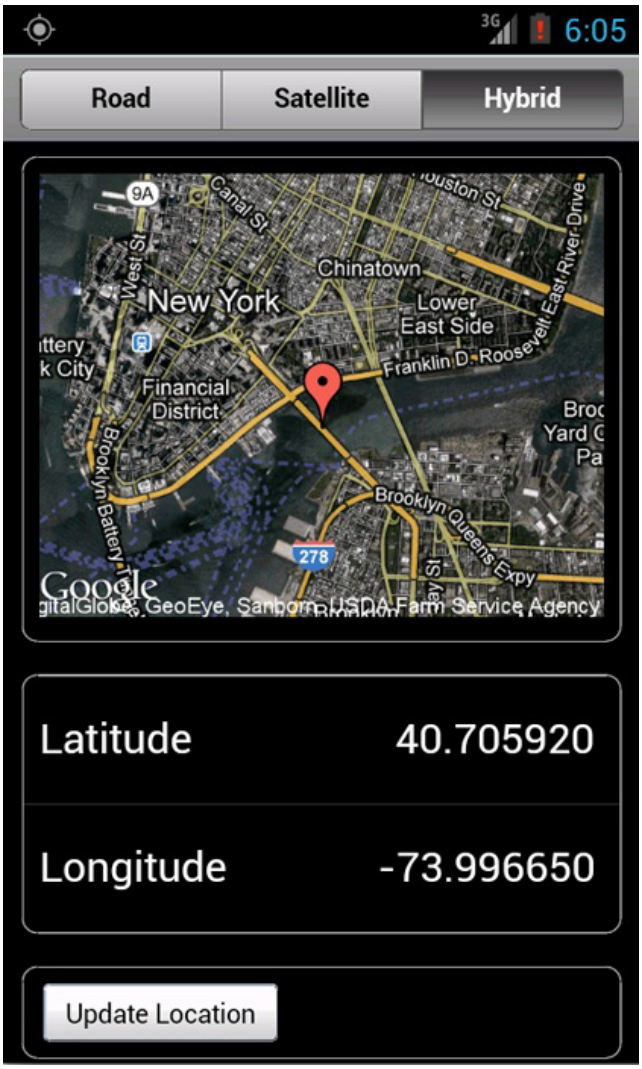
```
telnet localhost 5554
```

   If you're using Windows, note that `telnet` is likely not enabled by default.
2. Set the latitude and longitude using the `geo` command. For example, to set the location to the Brooklyn Bridge, the command would be:

```
geo fix -73.99665 40.705921
```

After setting the latitude and longitude you should see the map get centered on the Brooklyn Bridge, otherwise you will need to click the **Update Location** button. Figure 29 shows the app running in the emulator after the location update (and after clicking the **Hybrid** button to change the map type).

Figure 29. Locator app after position has been set



And, with that, you've successfully verified that the GPS locator works as a native Android app!

Revisiting the weight tracker

The steps we've taken to export the GPS locator from Maqetta, build with PhoneGap, and deploy to a mobile device/emulator would work with any Maqetta mobile prototype. As an example, Figure 30 shows the weight tracker prototype that we developed in Parts 1 and 2 running within the Android Emulator, after being built with PhoneGap.

Figure 30. Weight tracker in Android Emulator

Conclusion to the Maqetta series

In this three-part series, we've used Maqetta to quickly build prototypes for two different mobile apps. In the process, we've managed to cover a lot of ground, and hopefully provided a thorough introduction to Maqetta. For both apps, we started with a reasonable prototype that didn't require writing any code, then extended its interactive features by adding some custom JavaScript. In this last article, we combined Maqetta with PhoneGap, which we used to turn a GPS locator application prototype into a native app, which we then tested using the Android Emulator.

If you want to learn more about Maqetta, start with some of the links in the Resources section, and also read Tony's blog. Also keep in mind that you can use the step-by-step instructions in this article as a guide for developing future Maqetta mobile applications and deploying them with PhoneGap.

Acknowledgments

Special thanks to the Maqetta team (Jon Ferraiolo, Javier Pedemonte, Adam Peller, and Bill Reed) for thoughtfully reviewing and providing constructive feedback on this series of articles.

Download

Description	Name	Size
Final source of the custom app	maqetta_part3.zip	5KB

Resources

Learn

Read the Maqetta series:

Part 1: Design an HTML5 mobile UI (Tony Erwin, developerWorks, January 2013): Learn about Maqetta's major features while creating a prototype for a rich mobile application.

Part 2: Write custom JavaScript for your Maqetta mobile UI (Tony Erwin, developerWorks, February 2013): Take your prototype application to the next level by writing custom JavaScript to add interactive functionality.

Mobile application development, Part 1: PhoneGap and Dojo Mobile on Android (Bryce Curtis, Gill Woodcock, Todd Kaplinger, developerWorks, September 2011): A hands-on introduction to integrating PhoneGap and Dojo Mobile on Android.

Maqetta documentation: Get tutorials and cheat sheets from the Maqetta development team.

MaqettaYouTube Channel: Check out online video demonstrations, including an introduction to Maqetta composition types.

Tony Erwin's Maqetta blog: Learn more from this author, who is part of the Maqetta development team.

@Maqetta: Follow Maqetta on Twitter.

Maqetta on developerWorks: More resources for learning how to work with Maqetta.

HTML5 fundamentals: Follow this developerWorks knowledge path to learn the fundamentals of working with HTML5.

The W3C HTML5 Wiki: Learn even more about HTML5.

"Getting Started with dojox/mobile" (Dojo.org): Find out more about Dojo Mobile, a framework for creating cross-device-compatible mobile web applications.

"What's new in Dojo Mobile 1.8, Part 1: New widgets" (Yoshiroh Kamiyama, developerWorks, November 2012): Discover the new widgets, widget-related utility classes, and modules introduced in Dojo Mobile 1.8.

Google Static Maps API: Learn more about the Maps API used in this article.

In the developerWorks Mobile development site, access and learn how to use the latest tools and technologies for mobile application developers in the comprehensive IBM MobileFirst product portfolio. Explore our free software downloads and cloud trials, sample code, expert how-to advice, videos, training, and discussion—all focused on helping you develop, test, integrate, analyze, secure, and manage multi-platform mobile applications and deployments in your organization.

Get products and technologies

Download Maqetta: Install Maqetta on your own intranet server after

Dig deeper into Mobile development on developerWorks

Overview

Products

Learn about mobile technologies

Technical library (articles and more)

Connect

retrieving it from the downloads page or launch Maqetta in the cloud.

Download PhoneGap: A free and open source framework that makes it easy to create mobile apps using standardized web APIs.

Get the Ripple Emulator: A Google Chrome extension that provides a browser environment where a subset of the PhoneGap APIs are emulated.

Download the Android ADT Bundle: Combines the Android SDK, the Android Platform-tools, and a version of the Eclipse IDE with the ADT plugin.

Discuss

Join the Maqetta user group: Interact with other designers and developers using Maqetta to create desktop and mobile UIs.

Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.