Maqetta means mockup, Part 2: Write custom JavaScript for your Maqetta mobile UI

Develop an interactive UI prototype with JavaScript and the Dojo Toolkit

Tony Erwin
Software Engineer
IBM

Skill Level: Intermediate

Date: 11 Mar 2013

As you learned in Part 1 of this series, Maqetta is a WYSIWYG application that makes it easy to design a sophisticated desktop or mobile UI without writing any code. But what if you need a richer UI that responds to user input in more advanced ways? In this follow-up article, Tony Erwin walks you through the process of enhancing your Maqetta mobile UI with custom JavaScript using Dojo and the Dojo Mobile library.

View more content in this series

Introduction

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 this part, 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 in Tony's blog on developerWorks.

If you read the first article in this series, then you know that Maqetta is a browser-based application for designing and developing desktop and mobile UIs. That article showed you how to use Maqetta's drag-and-drop interface to design a rich mobile UI

prototype without writing any code. Though the prototype was live — meaning that a user could interact with its features — the data on the views was mostly static. While a prototype like this is often sufficient for proof-of-concept, some situations (such as trying to sell your concept to a client or potential investor) require a higher fidelity prototype that better demonstrates the application's behavior in practice.

In this article, we'll take our weight tracker application prototype to the next level with custom JavaScript. The updated application will leverage interactive features from Dojo and the Dojo Mobile library to respond to user events and dynamically change the data shown by widgets in each view.

If you haven't read Part 1 or previously developed a Maqetta UI prototype on your own, the examples could be difficult to follow. I recommend that you familiarize yourself with the fundamentals of Maqetta UI development before continuing.

A richer prototype

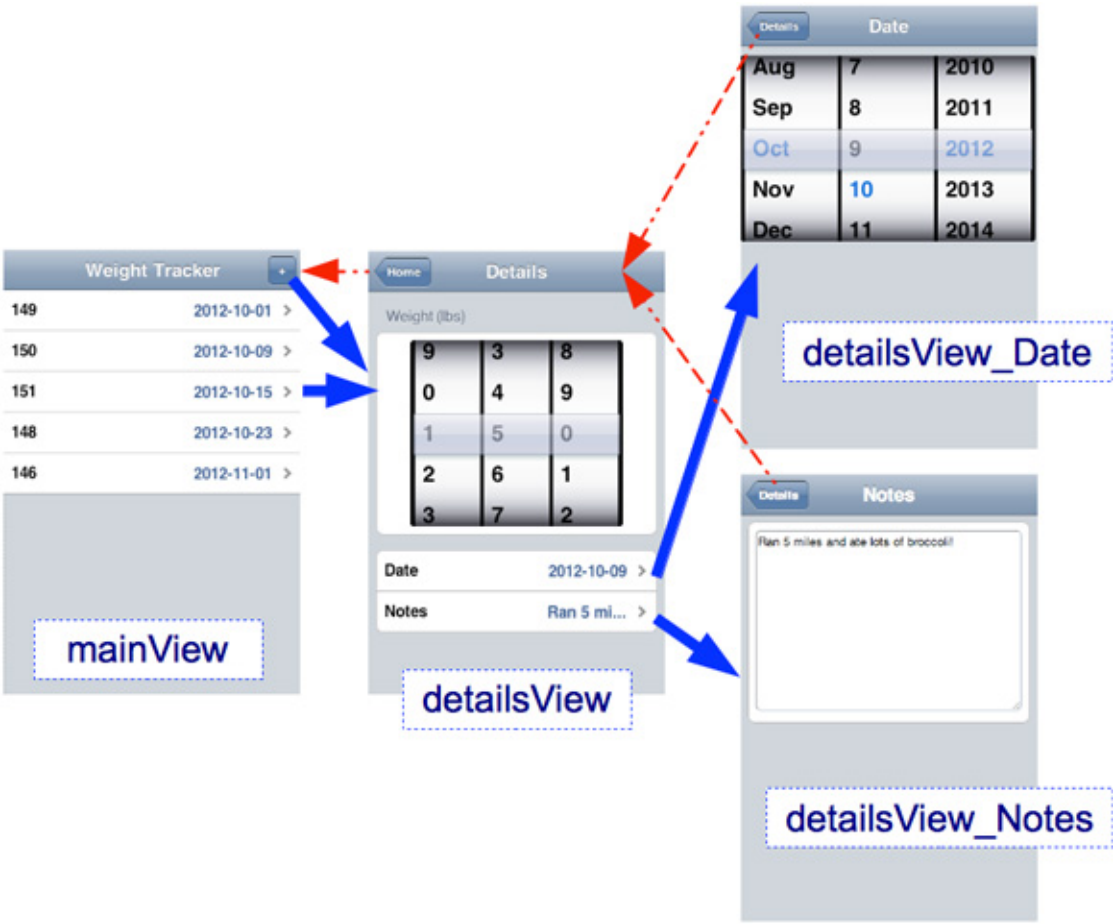
From Part 1, you may recall the weight tracker app flow (see Figure 1). When a user clicks a row in the weight list in `mainView`, he or she is taken to `detailsView` to see and edit more detailed information about that weight entry. While the prototype is visually rich and interactive, it is lacking in key areas:

- The data shown on `detailsView`, `detailsView_Date`, and `detailsView_Notes` is not based on the selected item in the weight list.
- Changes made to the data on those views are not reflected in the main weight list when a user navigates back to `mainView`.
- The plus (+) button on `mainView` does not actually add a new item to the weight list.

In this article, we'll address these shortcomings by adding custom JavaScript to the weight tracker application prototype. As a result, users of the prototype will have a richer, more realistic experience of the desired functionality.

Figure 1. A flowchart for the weight tracker application

Enlarge the figure

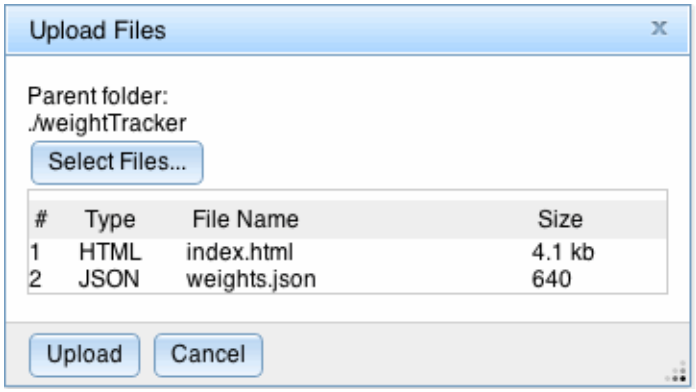


Set up your workspace

Examples in this article are based on the `index.html` and `weights.json` files we created for the weight tracker UI prototype in Part 1. Ideally, you're familiar with both of these files. If you've previously used Maqetta to design a mobile UI, and you want to focus on writing custom JavaScript for more dynamic UI features, then you can download the files from Part 1, unzip them, and upload them into your Maqetta workspace as follows:

1. In Maqetta's Files palette (on the lower left of the display), right-click one of the root-level files (such as `app.js`) and choose the **Upload files...** option.
2. In the Upload Files dialog, click **Select Files...**. An OS-specific dialog then invites you to choose files from your file system. Choose the files you want to upload (`index.html` and `weights.json`), and they will be added to the list in the Upload Files dialog, as shown in Figure 2.

Figure 2. Uploading files



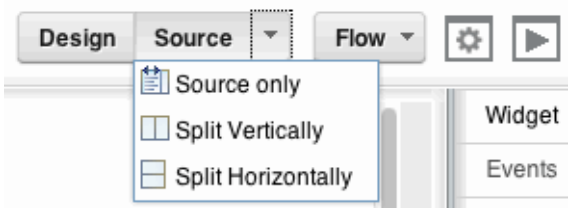
3. Click the **Upload** button, and the files will be uploaded and will appear in your Files palette. You can use these files just like any that you had created yourself, directly in Maqetta.

Maqetta's HTML source

Before writing JavaScript for a Maqetta prototype, you should be familiar with the prototype's generated HTML source. Let's view the source for `index.html`:

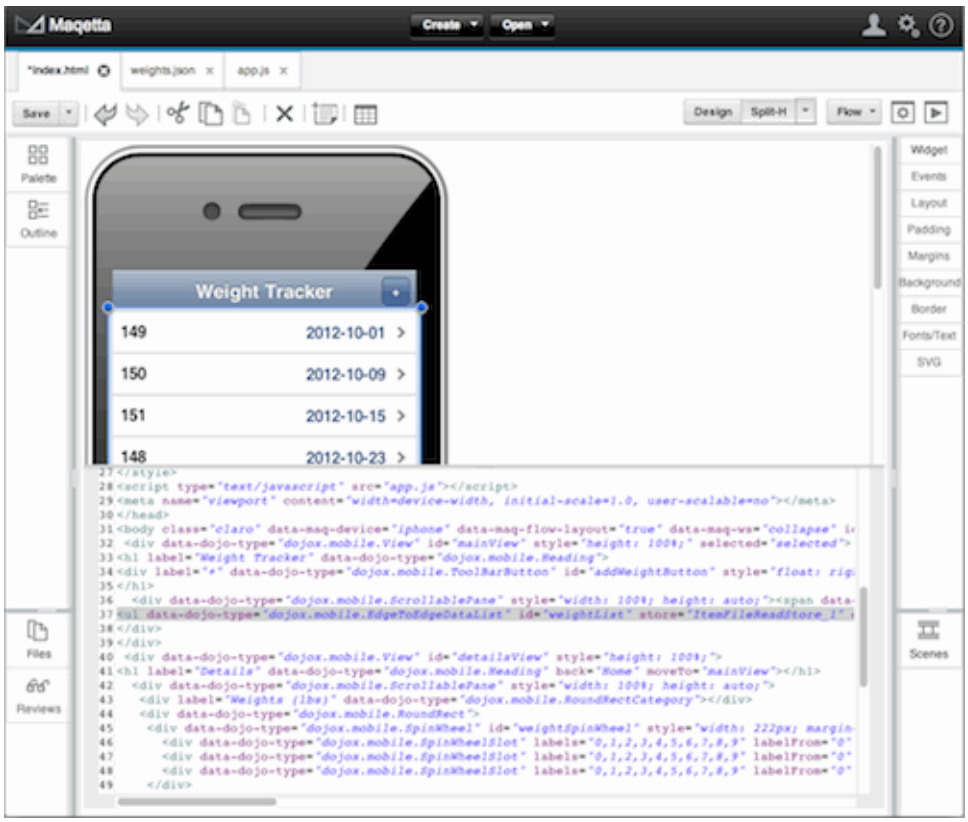
1. Open `index.html` in the Maqetta page editor.
2. If you want to see just the source (and not your design canvas), click the **Source** button in the Maqetta toolbar. Click **Design** to get your design canvas back.
3. If you want to see the source for the file alongside your design canvas, open the pull-down menu next to the **Source** button (as shown in Figure 3) and choose **Split Vertically** or **Split Horizontally**. The option you choose will become the default view the next time you click the **Source** button.

Figure 3. View options in the Source menu



Note that when you're working in a split screen, you can select a widget in the design pane to see its source highlighted in the source pane. In Figure 4, I've selected the `EdgeToEdgeDataList` from the `mainView`, and its HTML is highlighted in the source pane:

Figure 4. The horizontal source pane



Explore the HTML source

One of the first things that the Maqetta-generated HTML source does is to set up Dojo. When the HTML is loaded, the `dojo/parser` (see Resources for more information) parses the HTML tags in the body of the document and creates Dojo widgets as indicated. These Dojo widgets are JavaScript objects, which we can reference and manipulate with custom JavaScript.

The Dojo Reference Guide

The rich mobile application UI developed in this article uses functionality and widgets found in the Dojo, Dijit, and Dojo Mobile (`dojox/mobile`) packages of the Dojo Toolkit. Please refer to the Dojo Reference Guide (1.8 as of this writing) to learn more about these components.

Listing 1 shows the HTML snippet from `index.html` that makes up the `mainView`. The very first line defines an HTML `<div>` element. Note that the value for the `data-dojo-type` attribute tells the Dojo parser to create an instance of `dojox/mobile/ScrollableView`, which is a JavaScript class from the Dojo Mobile library (see Resources). It represents the `ScrollableView` widget in our `mainView`.

Also note that the `ID` is set to `mainView` (you might recall setting this `ID` in Part 1, using the Properties palette). Knowing a Dojo widget's type and `ID` makes it relatively straightforward to write JavaScript to change the widget's runtime behavior, as you'll see shortly.

Listing 1. Snippet for mainView from index.html

```
<div data-dojo-type="dojox.mobile.ScrollableView" id="mainView"
        keepScrollPos="false" scrollBar="true" selected="selected">
    <h1 label="Weight Tracker" data-dojo-type="dojox.mobile.Heading" fixed="top">
        <div label="+" data-dojo-type="dojox.mobile.ToolBarButton"
                moveTo="detailsView" style="float: right;"></div>
    </h1>
    <span data-dojo-type="dojo.data.ItemFileReadStore" id="ItemFileReadStore_1"
            jsId="ItemFileReadStore_1" url="weights.json"></span>
    <ul data-dojo-type="dojox.mobile.EdgeToEdgeDataList" store="ItemFileReadStore_1"
            query="{"label":"*"}"></ul>
</div>
```

Other elements in Listing 1 should have familiar attributes. Inside the `<div>` element for `mainView`, for instance, we see the following:

- An `<h1>` element of Dojo type `dojox/mobile/Heading` with a label of "Weight Tracker." Inside this element is a `<div>` element of Dojo type `dojox/mobile/ToolBarButton` with a label of "+" and an `ID` of `addWeightButton`. We'll write some JavaScript that responds to a click event on this button by adding a new entry to the weight list.
- A `<span>` element of Dojo type `dojo/data/ItemFileReadStore` that specifies a URL of `weights.json`.
- A `<ul>` element of Dojo type `dojox/mobile/EdgeToEdgeDataList` with an `ID` of `weightList` and a `store` attribute. Note that the `store` attribute points to the `ItemFileReadStore` that was created on the previous line. We'll write JavaScript that gets a reference to this list and modifies its data store.

Update weights.json

Before we start writing custom JavaScript, we need to add some new and updated fields to `weights.json`. Open `weights.json` by double-clicking it in the Files palette, which sits on the lower left of the Maqetta workbench. Then replace the contents with what you see in Listing 2 and save the file.

Listing 2. Update to weights.json

```
{
    "identifier": "id",
    "items": [
        {id: "weight_0", label: "149", moveTo: "#", rightText: "2012-10-01",
  notes: "Starting to track my weight."},

        {id: "weight_1", label: "150", moveTo: "#", rightText: "2012-10-09",
  notes: "Ran 5 miles and ate lots of broccoli!"},

        {id: "weight_2", label: "151", moveTo: "#", rightText: "2012-10-15",
  notes: "Oops, going in wrong direction."},

        {id: "weight_3", label: "148", moveTo: "#", rightText: "2012-10-23",
  notes: "Wow, lost 3 pounds!"},

        {id: "weight_4", label: "146", moveTo: "#", rightText: "2012-11-01",
   notes: "Feeling good!"}
    ]
}
```

Each line in Listing 2 contains an item meant to represent a weight entry. Each item now contains some new and changed attributes:

- `ID` is a new field that acts as the unique identifier for a given weight (note the newly added line `"identifier": "ID"`). This field will make it easier for us to look up and modify individual items in JavaScript.
- `label` is an existing field that holds the weight of a given item.
- `moveTo` is an existing field that I've changed to `"#"`, which will enable us to handle the transition to `detailsView` in JavaScript.
- `rightText` is an existing field that holds the date of an item.
- `notes` is a new field that holds a text note for any given item.

Because this article is about JavaScript, I should point out that `weights.json` conforms to the *item structure* (see Resources) of `ItemFileReadStore`. Recall from looking at the HTML source that this is the kind of data store used by the `EdgeToEdgeDataList`.

JavaScript in Maqetta

For the remainder of this article, we'll be adding custom JavaScript to the `app.js` file for our weight tracker application. Note that an `app.js` is provided by default for all Maqetta projects. If you look at the source for `index.html` (or any Maqetta-generated HTML file), you will see that it loads `app.js` via the line below:

```
<script type="text/javascript" src="app.js"></script>
```

Start by opening the `app.js` file in your project, which you can do by double-clicking it in the Files palette. The file should look something like what you see in Listing 3.

Listing 3. Default app.js file

```
/*
 * 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"], function(ready){
    ready(function(){
        // logic that requires that Dojo is fully initialized should go here

    });
});
```

Note that the `app.js` file uses `dojo/ready`. Any code placed within the `ready` function is guaranteed to run only after the necessary Dojo resources have been fully loaded, the page has been parsed, and the specified Dojo widgets have been created. (See Resources to learn more about `dojo/ready`.)

Simple app.js demo

To get your feet wet with `app.js`, let's add an alert message to the file that will display when the application loads in a browser:

1. Add a simple alert statement to `app.js` like the one shown in Listing 4:
   Listing 4. An alert added to app.js
   ```
   /*
    * 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"], function(ready){
        ready(function(){
            // logic that requires that Dojo is fully initialized should go here

            //Add a temporary alert just to make sure we're working
            alert("Code from app.js is running!");
        });
   });
   ```
2. Save `app.js`.
3. Preview the `index.html` file with the **Preview in Browser** button, and note that the browser shows an alert dialog after the weight tracker app has loaded.

Add custom JavaScript

Most of our work to enhance the weight tracker application will happen in `app.js`. We'll add custom JavaScript to this file to call and manipulate various Dojo widgets. You can follow along by copying and pasting JavaScript code snippets into your `app.js` file as they are presented.

Many of the JavaScript snippets represent testable enhancements to the app. As you add code, you can use Maqetta's Preview function to test the new features. Or, if you prefer to skip ahead, simply download the final version of `app.js` and replace your current version with that one.

Required modules

The default `app.js` file uses the `require` function defined by the Dojo loader to load a single Dojo module (the previously mentioned `dojo/ready`). For our purposes, we're going to need a few more modules, including the following (see the Dojo Toolkit Reference Guide in Resources):

- `dojo/dom`
- `dojo/dom-style`
- `dijit/registry`
- `dojo/on`
- `dojo/date/stamp`
- `dojo/data/ItemFileWriteStore`

To add these modules to your `app.js`, simply replace it with the code in Listing 5, which includes the additional modules.

Listing 5. Additional Dojo modules for app.js

```
/*
 * 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/date/stamp",
        "dojo/data/ItemFileWriteStore"],
function(ready,
         dom,
         domStyle,
         registry,
         on,
         stamp,
         ItemFileWriteStore){

    ready(function(){
        // logic that requires that Dojo is fully initialized should go here

    });
});
```

Referencing widgets

Our next task is to use JavaScript to get a reference to every Dojo widget that we want to interact with. Recall from looking at the generated HTML that we can get a reference to any Dojo widget whose `ID` we know. This is especially easy with the help of the `byId` function, which is found in the `dijit/registry` module. For instance, we would access the weight-list widget like so: `registry.byId("weightList")`.

Listing 6 calls `registry.byId` a number of times, each time looking up an individual widget and storing its reference in a variable that we'll use later. As a safeguard, I've added an `if` statement to ensure that all of the variables have been defined (note that it's easy to mistype or forget to enter an `ID`). In the case of a missing variable, an error message will appear to help us track down the problem.

Listing 6. Getting a reference to widgets

```
        /* ****************************************
         * Get a reference to all the widgets we need
         ******************************************/
        var weightList = registry.byId("weightList");
        var mainView = registry.byId("mainView");
        var detailsView = registry.byId("detailsView");
        var detailsView_Date = registry.byId("detailsView_Date");
        var detailsView_Notes = registry.byId("detailsView_Notes");
        var weightSpinWheel = registry.byId("weightSpinWheel");
        var dateListItem = registry.byId("dateListItem");
        var notesListItem = registry.byId("notesListItem");
        var dateSpinWheel = registry.byId("dateSpinWheel");
        var notesTextArea = registry.byId("notesTextArea");
        var addWeightButton = registry.byId("addWeightButton");

        // Make sure we found all of the widgets
        if (!weightList ||
            !mainView ||
            !detailsView ||
            !detailsView_Date ||
            !detailsView_Notes ||
            !weightSpinWheel ||
```

```
            !dateListItem ||
            !notesListItem ||
            !dateSpinWheel ||
            !notesTextArea ||
            !addWeightButton) {

            // show 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 weightList = " +  weightList + ",\n" +
                "\t mainView = " +  mainView + ",\n" +
                "\t detailsView = " +  detailsView + ",\n" +
                "\t detailsView_Date = " +  detailsView_Date + ",\n" +
                "\t detailsView_Notes = " +  detailsView_Notes + ",\n" +
                "\t weightSpinWheel = " +  weightSpinWheel + ",\n" +
                "\t dateListItem  = " +  dateListItem + ",\n" +
                "\t notesListItem = " +  notesListItem + ",\n" +
                "\t dateSpinWheel = " +  dateSpinWheel + ",\n" +
                "\t notesTextArea = " +  notesTextArea + ",\n" +
                "\t addWeightButton = " +  addWeightButton);

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

Go ahead and copy and paste this code immediately inside the `ready` function in your `app.js` file. Then save the file and preview the app to ensure that all of the widgets are there.

Change the data store

You saw earlier in the HTML source that Maqetta generated an `ItemFileReadStore` to be used by our `EdgeToEdgeDataList` in the `mainView`. While the `ItemFileReadStore` is good for static data, we now want to modify the weight tracker application's data at runtime, so we need our `EdgeToEdgeDataList` to use an `ItemFileWriteStore`.

`ItemFileWriteStore` has all the functionality of an `ItemFileReadStore`, but it adds implemented functions required by the `dojo/data/api/Write` and `dojo/data/api/Notification` APIs. Because of these additions, we'll be able to modify data in this datastore with some specific function calls. Listing 7 shows the code to update the data store used by the weight list.

Listing 7. Set a different data store

```
        /* ****************************************
         * Replace ItemFileReadStore generated by
         * Maqetta with ItemFileWriteStore
         ******************************************/
        var weightWriteStore = new ItemFileWriteStore  ({
            url:"weights.json"
        });
        weightList.setStore (weightWriteStore);
```

Copy the code from Listing 7 and paste it in to your `app.js` file after the `if` statement that we added in Listing 6. After saving the `app.js` file, try previewing the weight tracker app. You should see that the weight list from your `weights.json` file is still displayed in the `EdgeToEdgeDataList`.

Add a placeholder for selected data

Our general strategy is to store the data for a currently selected weight entry in the `selectedWeightData` variable. The code in Listing 8 defines that variable. Add the code below to your `app.js` file. Note that you won't see any difference in functionality if you preview the weight tracker app after making the change.

Listing 8. Placeholder for selected data

```
/* ****************************************
 * Provide placeholder for the weight data
 * currently being edited.
 ****************************************/
var selectedWeightData = null;
```

Add a click handler

Next we'll define a function called `listItemClick`, which will eventually be invoked every time an item in `mainView`'s `EdgeToEdgeDataList` is clicked. The `listItemClick` function expects a `dojoListItem` parameter of type `dojox/mobile/ListItem`. We'll use the reference to the `ListItem` to populate the `selectedWeightData` variable from Listing 8.

After we've populated `selectedWeightData`, we'll initiate the transition to `detailsView`. (Recall that in Listing 2 we changed `moveTo` to `"#"` so that we could manually handle this transition.)

Listing 9 shows the JavaScript to define `listItemClick`.

Listing 9. Click handler

```
/* ****************************************
 * Function to be called when item in the
 * EdgeToEdgeDataList is clicked.
 ****************************************/
var listItemClick = function(dojoListItem) {
    // Fill in selected weight data based on selected item
    selectedWeightData = {
        id: dojoListItem.params.id,
        label: dojoListItem.params.label,
        rightText: dojoListItem.params.rightText,
        notes: dojoListItem.params.notes,
    };

    //Perform the transition
    dojoListItem.transitionTo("detailsView");
};
```

Go ahead and add the new click handler to your `app.js` now. You won't see any changes in the application preview until we attach it in the next section.

Listen for the click

Next we want the `listItemClick` function to be called whenever a weight-list item is selected, but there's no direct way to add a click handler to individual list items. You might recall, though, that the `EdgeToEdgeDataList` was represented in the HTML source by a `<ul>` element. At runtime, the `EdgeToEdgeDataList` will create individual

`<li>` elements for each item displayed in the list. We'll use that knowledge to create the weight-list functionality that we want.

In Listing 10, we use `dojo/on` to register a function that will be called anytime the `EdgeToEdgeDataList` is clicked. We then examine the target of the event to determine what sub-element of the list was underneath the mouse when a click happened. That sub-element will be a descendant of the `<li>` element that we're interested in. We search the sub-element's ancestry to find the `<li>`. Once we've found the `<li>`, we can use `registry.byId` to get a reference to the Dojo `ListItem` and call `listItemClick`.

Listing 10. Click handler for EdgeToEdgeDataList

```
/* ****************************************
  * When weight list is clicked, we want to
  * find the Dojo ListItem that was actually
  * targeted by the user and handle the click.
  ****************************************/
 on(weightList, "click",
     function(event) {
         // The event's "target" will be the list's
         // sub-element what was clicked. (The
         // event's "currentTarget" should be the list
         // itself.
         var subElement = event.target;

         // The subElement of the list may be an LI or
         // a child of an LI element. If not an LI,
         // we want to search the ancestry of the
         // subElement to find the LI.
         var parent = subElement.parentNode;
         while (parent != null && parent.nodeName != "LI") {
             parent = parent.parentNode;
         }

         if (parent) {
             // If parent is set, then we've found the LI. From
             // there we can use the id to get the Dojo ListItem.
             var dojoListItem = registry.byId(parent.id);

             // Handle the click
             listItemClick(dojoListItem);
         }
 });
```

After updating `app.js` with this code and saving it, preview the weight tracker application. Does clicking an item in the `EdgeToEdgeDataList` cause a transition to `detailsView`? Recall that the transition is now initiated by the `listItemClick` function. So, if the `listItemClick` function is executed, the transition should occur. And, if the transition occurs, we can also be reasonably certain that `selectedWeightData` is being populated with the data from the selected list item.

Monitor transition events

At this point we've set up a good base in `app.js` to implement the functionality we care about, and we're done with the hardest part of adding more dynamic functionality to our UI prototype. For the next several sections, we'll repeat the

same general strategy to monitor the transition events that occur as various views are shown and hidden. In the process, you'll get more familiar with adding custom JavaScript to a Maqetta UI prototype.

Details view transitions

Just before a view becomes visible (as indicated by an `onBeforeTransitionIn` event), we want to set the values on the widgets in that view based on the data in `selectedWeightData`. Likewise, just before a view is hidden (as indicated by an `onBeforeTransitionOut` event), we will update the data in the `selectedWeightData` variable. The update will reflect any changes that the user has made while interacting with the UI widgets in the view. After we update the `selectedWeightData`, any changes will be available to the transition handler for the next view to be shown.

In Listing 11 below, we start by exploring the transition into `detailsView`. Using the `dojo/on` module, we specify a function to be called when the `beforeTransitionIn` event is fired for `detailsView`. In that function, we update `weightSpinWheel`, `dateListItem`, and `notesListItem` based on the values we find in `selectedWeightData`. Note the following about the code in Listing 11:

- Updating `weightSpinWheel` requires some maneuvering because we need to set the value for each each spin-wheel slot based on the digits in the weight (which is located in `selectedWeightData.label`). It happens that `weightSpinWheel` is an instance of `dojox/mobile/SpinWheel`, so we can get the slots by calling the `getSlots` function. Each slot is an instance of `dojox/mobile/SpinWheelSlot`. Like most Dojo Mobile widgets, we can set its value attribute via the `set` function.
- We update the `rightText` attribute for `dateListItem` by issuing a call to its `set` function. We also use `set` to update the `rightText` attribute for `notesListItem`.
- After doing the update, we'll use `domStyle.get` to get the width of the `DOM` node holding the right-text for the `dateListItem`. Then we'll use that value with `domStyle.set` to change the width of the `DOM` node holding the right-text for the `notesListItem`. With the width set properly, we can use other CSS attributes (namely `white-space`, `overflow`, and `text-overflow`) to cause the `rightText` to be truncated (with an ellipsis) if necessary.

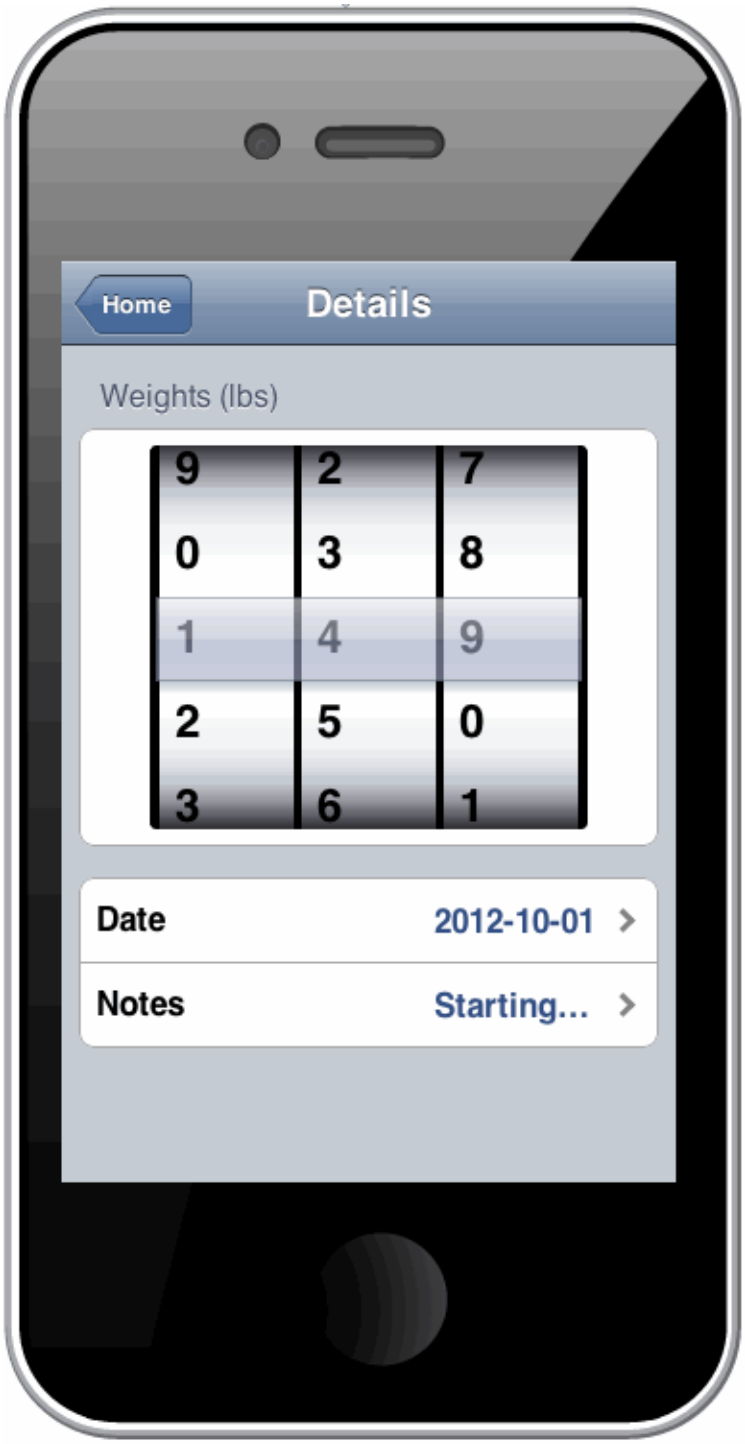
Listing 11. Before transitioning into the Details view

```
/* ****************************************
 * detailsView Transitions
 ****************************************/
on(detailsView, "beforeTransitionIn",
    function(){
      if (selectedWeightData) {
        // Get the slots from the spin wheel
        var weightSpinWheelSlots = weightSpinWheel.getSlots();

        // Loop over digits in weight to set value for each slot in the
        // spin wheel. For simplicity (and this is a prototype
        // after all) assuming all weight labels have a string length
        // of 3 (e.g., weight > 100)
        for (var i = 0; i < 3; i++) {
           var char = selectedWeightData.label.charAt(i);
```

```
                weightSpinWheelSlots[i].set("value", char);
            }

            // Update the date list item
            dateListItem.set("rightText", selectedWeightData.rightText);

            // Update the notes list item
            notesListItem.set("rightText", selectedWeightData.notes);

            // Update styling of notesListItem's rightTextNode so that
            // it's the same width as the date label and will automatically
            // add an ellipsis for us. The settings for whiteSpace,
            // overflow, and textOverflow are static, so they technically
            // should go in app.css and override the "mblListItemRightText"
            // style class.
            var width = Math.round(domStyle.get(dateListItem.rightTextNode, "width"));
            domStyle.set(notesListItem.rightTextNode, "width", width + "px");
            domStyle.set(notesListItem.rightTextNode, "whiteSpace", "nowrap");
            domStyle.set(notesListItem.rightTextNode, "overflow", "hidden");
            domStyle.set(notesListItem.rightTextNode, "textOverflow", "ellipsis");
        }
    });
```

After updating `app.js` with the code snippet above, our application preview should start behaving a lot more like we want it to. If you click an item in `mainView`, the widgets in `detailsView` should actually reflect the item clicked. For example, Figure 5 shows the result of clicking on the first item in the `EdgeToEdgeDataList` in `detailsView`. Note how the values are different from the default values you see in the Maqetta page editor. Also note that if you go back to the `mainView` (by clicking the **Home** button in the heading), and select a different item in the weight list, that the `detailsView` again has different values!

Figure 5. Details view with non-default values



Transition out of detailsView

Now we'll deal with what happens when the user leaves the `detailsView`. Recall that when we leave a view, we want to update `selectedWeightData` to reflect any changes that the user made while in that view. In Listing 12 we use `dojo/on` to register a function to be called when a `beforeTransitionOut` event is fired for `detailsView`.

The only widget that the user can edit in this view is the `weightSpinWheel`, so `weight` (stored in `selectedWeightData.label`) is the only thing we modify when leaving the view. We again use the `getSlots` function to get the individual slots, but this time we loop over the slots, pulling the value attribute from each one and concatenating the results to form a string that represents the weight. We then put this value back in `selectedWeightData`.

Listing 12. Before transitioning out of the Details view

```
        on(detailsView, "beforeTransitionOut",
            function(){
                if (selectedWeightData) {
                    // Get the slots from the spin wheel
                    var weightSpinWheelSlots = weightSpinWheel.getSlots();

                    // Build up the new label for weight from the weight spin wheel slots
                    var newLabel = "";
                    for (var i = 0; i < weightSpinWheelSlots.length; i++) {
                        newLabel += weightSpinWheelSlots[i].get("value");
                    }

                    // Update selected weight data
                    selectedWeightData.label = newLabel;
                }
        });
```

When you update `app.js` with the snippet above, you won't notice any difference in the behavior of the preview. Soon, however, we'll implement a `beforeTransitionIn` event handler for `mainView` that will take advantage of the updated value in `selectedWeightData.label` to update the list item. Then you'll start to see some changes!

Transitions for detailsView_Date

With `app.js` in its current state, the widgets in `detailsView` reflect the values in the weight item that was clicked in the `mainView`. But, if you clicked the `dateListItem` in `detailsView`, to get to `detailsView_Date`, you would see that the date spin-wheel is still populated with the static value set in the Maqetta page editor, and not the runtime selection. So we need to implement some more transition handlers.

In the Listing 13, we register a function to run every time the `beforeTransitionIn` event is fired for `detailsView_Date`. In this case, the implementation is very simple. We just set the value attribute on the `dateSpinWheel` (an instance of `dojox/mobile/SpinWheelDatePicker`) to `selectedWeightData.rightText`. Note that this works because we've been representing our dates using the ISO-8601 format that `SpinWheelDatePicker` works with.

Listing 13. Before transition in for detailsView_Date

```
/* ****************************************
 * detailsView_Date Transitions
 ****************************************/
on(detailsView_Date, "beforeTransitionIn",
    function(){
        if (selectedWeightData) {
            // NOTE: Date spin wheel expects an ISO date (which is
            // what we've been putting in rightText)
            dateSpinWheel.set("value", selectedWeightData.rightText);
        }
});
```

If you run the Preview function again after updating the `app.js` file with the code above, you should find `detailsView_Date` working how we want it to. After picking a weight item on `mainView`, the value in the date spin-wheel will match the `rightText` of both the item in the `EdgeToEdgeDataList` and the date-list item in `detailsView`.

The only thing left to do with `detailsView_Date` is to update `selectedWeightData` with the value of the date spin-wheel, in case the user has changed it.
As Listing 14 shows, when the `beforeTransitionOut` event is fired for `detailsView_Date`, we simply get the value from `dateSpinWheel` and then set `selectedWeightData.rightText`.

Listing 14. Before transition out for detailsView_Date

```
on(detailsView_Date, "beforeTransitionOut",
    function(){
        if (selectedWeightData) {
            // Get value from the spint wheel
            var value = dateSpinWheel.get("value");

            // Update selected weight data
            selectedWeightData.rightText = value;
        }
});
```

Update `app.js` and run Preview. If you change the value in the date spin-wheel on `detailsView_Date`, then when you go back to `detailsView`, the `rightText` of the `dateListItem` will be updated with the new value.

Transitions for detailsView_Notes

We'll follow the same basic process from `detailsView_Date` to update the code for `detailsView_Notes`. As Listing 15 shows, the handlers for `beforeTransitionIn` and `beforeTransitionOut` for `detailsView_Notes` are very similar to the respective `detailsView_Date` handlers. The difference is that `notesTextArea` and `selectedWeightData.notes` are used instead of `dateSpinWheel` and `selectedWeightData.rightText`.

Listing 15. Transitions for detailsView_Notes

```
/* ****************************************
 * detailsView_Notes Transitions
 ****************************************/
```

```
        on(detailsView_Notes, "beforeTransitionIn",
            function(){
                if (selectedWeightData) {
                    notesTextArea.set("value", selectedWeightData.notes);
                }
        });

        on(detailsView_Notes, "beforeTransitionOut",
            function(){
                if (selectedWeightData) {
                    // Get value from the text area
                    var value = notesTextArea.get("value");

                    // Update selected weight data
                    selectedWeightData.notes = value;
                }
        });
```

After updating `app.js` with the code above, re-run Preview. You should find that the `notesTextArea` in `detailsView_Notes` reflects the value of the `notes` attribute in the item that was selected in `mainView`. And, if you make changes at the start of the string in `notesTextArea` and go back to `detailsView`, then the `rightText` for the `notesListItem` should also be updated.

Main view transitions

All of the transition code is in place for `detailsView`, `detailsView_Date`, and `detailsView_Notes`. Next we want to ensure that our `EdgeToEdgeDataList` shows the most current data when a user transitions into `mainView` from `detailsView`. For this, we will mostly be interacting with the `weightWriteStore`. Recall that we created `weightWriteStore` early in our updates of `app.js` (see Listing 7). It is an instance of `ItemFileWriteStore`, whose task is to provide data to `weightList`.

Now we want to change values on the data item backing the list item that was originally clicked. So, in Listing 16, we start by invoking `fetchItemByIdentity` on the `weightWriteStore` with the `ID` of the item we're interested in. The `fetchItemByIdentity` function works asynchronously, so we need to pass it a function to invoke when the item has been fetched (via the `onItem` argument).

Once our `onItem` function is called, the process becomes more straightforward. We "commit" data changes by calling `setValue` on `weightWriteStore` three times to update `label`, `rightText`, and `notes` on the fetched item using the values stored in `selectedWeightData`. We then call `setStore` on `weightList` using `weightWriteStore` to cause the `EdgeToEdgeDataList` to refresh itself from the store. Finally, we null-out `selectedWeightData` because we no longer have an active selection.

Listing 16. Before transition in for mainView

```
        /* ****************************************
         * mainView transition
         ****************************************/
        on(mainView, "beforeTransitionIn",
            function(){
                if (selectedWeightData) {
```

```
                    weightWriteStore.fetchItemByIdentity({
                        identity: selectedWeightData.id,
                        onItem: function(item) {
                            // We've retrieved the item we want to edit, so
                            // update it in the weight list data store
                            weightWriteStore.setValue(item, "label",
                                selectedWeightData.label);
                            weightWriteStore.setValue(item, "rightText",
                                selectedWeightData.rightText);
                            weightWriteStore.setValue(item, "notes",
                                selectedWeightData.notes);

                            // Force weight list to reload the data store
                            weightList.setStore(null);
                            weightList.setStore(weightWriteStore);

                            //Clear out the selected weight data
                            selectedWeightData = null;
                        },
                        onError: function(error) {
                            // TODO: in production environment, would want to do
                            // something with error
                            console.error("fetchItemByIdentity failed!");
                        }
                    });
                }
        });
```

After updating `app.js` and previewing the app, you should see that we have accomplished most of our goals. In particular, changes made in `detailsView` and `detailsView_Date` are reflected in the `EdgeToEdgeDataList` after navigating back to `mainView`. Changes made in `detailsView_Notes` are less readily apparent, but we saw in the code above that we're committing the changes made to `notes` back to the `weightWriteStore`. To test this, re-select the same weight entry in `weightList` and navigate to `detailsView_Notes`. If you modified `notesTextArea` before, then you should now see your updated value.

Add weight entries

We have one last piece of JavaScript to write for our dynamic UI prototype. Recall that we wanted the plus (+) button in the `mainView` heading to add a new item to the list of weights, then transition to `detailsView` to allow the user to edit the new entry.

To do this, as shown in Listing 17, we first use `dojo/on` to register a function to be called when the `addWeightButton` is clicked. Within that function, we do the following:

1. Generate a unique `ID` for the new item using our `addWeightCounter` counter.
2. Use the unique `ID` to create default data for the new item.
3. Act like the new item was clicked and place its data in `selectedWeightData`. When the transition to `detailsView` occurs, its `beforeTransitionIn` handler will be able to use the data.
4. Call `newItem` on `weightWriteStore` with the data for the new item. Aside from adding to the data store, this fires appropriate events so that the `EdgeToEdgeDataList` will automatically create a new list-item widget representing the new weight.

Listing 17 shows the JavaScript for the plus button.

Listing 17. Handle click on addWeightButton

```
/* ****************************************
 * Handle addWeightButton
 ****************************************/
var addWeightCounter = 0;
on(addWeightButton, "click", function() {
    // Generate a unique id for the new item
    var newWeightId = "newWeight_" + addWeightCounter++;

    // Fill in some default data for the new item
    var newWeightData = {
        id: newWeightId,
        moveTo: "detailsView",
        //Default to 150, but production code would use most recent weight
        label: "150",
        //Default rightText to today's date
        rightText: stamp.toISOString(new Date(), {selector: 'date'}),
        //Default notes to empty string
        notes: ""
    };

    // Set the selected weight data to data for new item
    selectedWeightData = newWeightData;

    // Add new item to the data store. NOTE: We're keeping this simple for
    // the prototype and just always adding the new item to the data store.
    // That is, we're not considering possibility of user canceling the
    // operation.
    weightWriteStore.newItem(newWeightData);
});
```

When you update `app.js` and preview the application, the plus button should transition to `detailsView` and show data for the new item. If you go immediately back to `mainView`, you will see a new entry with a label of 150 and right-text with today's date. As one last test, try going back to `mainView` and changing the weight and/or date. When you go back to `mainView`, thanks to all of our transition handlers, the new entry should have those updated values.

Conclusion to Part 2

The Maqetta UI prototype that we developed in Part 1 was easy to build without writing any code and already pretty rich, but in this article, we've polished it up considerably. Adding custom JavaScript and using interactive features from Dojo and Dojo Mobile have improved the realism of the weight tracker's flow. The app is still far from complete (for instance, added or edited weights don't persist across sessions, and there's no way for a user to delete a weight entry), but that's fine for a prototype. The added features would amply demonstrate to an executive or potential investor how we intended for our UI to behave in practice.

Your experience with Maqetta after completing the exercises in Part 1 and Part 2 has prepared you to start building your own Maqetta prototypes, and even writing some custom JavaScript should you need to. In Part 3, the final article in this series, we'll continue to build on the concepts we've explored so far. First we'll build a quick

prototype for a GPS locator app, and then we'll use PhoneGap to turn that prototype into a native application to be deployed on actual mobile devices. In the meantime, see the Resources section to learn more about Maqetta.

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.

Downloads

Description	Name	Size	Download method
Final source of the custom app	maqetta_part2.zip	5KB	HTTP

Information about download methods

Resources

Learn

- "Maqetta means mockup, Part 1: Design an HTML5 mobile UI" (developerWorks, January 2013): Learn about Maqetta's major features while creating a prototype for a rich mobile application.
- Dojo Reference Guide (1.8 as of this writing): Learn more about the Dojo components and features used to develop a more interactive weight tracker application.
- Maqetta documentation: Get tutorials and cheat sheets from the Maqetta development team.
- Maqetta YouTube 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" (developerWorks, November 2012): Discover the new widgets, widget-related utility classes, and modules introduced in Dojo Mobile 1.8.

Get products and technologies

- The rich Maqetta mobile application UI developed in Part 2 uses functionality and widgets found in the Dojo, Dijit, and Dojo Mobile packages of the Dojo Toolkit.
- Maqetta.org: Launch Maqetta in the cloud.
- Download Maqetta: Install Maqetta on your own intranet server after retrieving it from the downloads page.

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.

About the author

Tony Erwin

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.