jQuery Mobile Applications

In this chapter, we'll look into some of the details of using jQuery and jQuery mobile as a framework for creating mobile apps. While it is totally possible to build a mobile web app using nothing more than the Javascript the we expect to be built into any modern browser, it turns out that such an application would not look at all like the type of program a user would expect to find on their phone; the user interface would not be well suited to an application on a small screen, page mark-up would tend to be more complex and the usual interactions a user would expect to work with – taps, swipes, selecting from a 'finger-sized' list etc. would not be built into the app.

jQuery is an add-on library for Javascript that did a lot to promote the creation of web-apps that worked in a similar way to desktop applications. jQuery Mobile extends that brief to provide a 'native' mobile look and feel that users are more likely to be familiar with.

jQuery Mobile App Framework

In software development terms a 'framework' is a reusable set of libraries, classes or objects. Typical frameworks provide common application components (e.g. databases, or a user-interfaces) so that it is easier to use these while building an application than it would be to start from scratch.

In the case of jQuery Mobile (jQM), the framework does not do anything you could not do by building your own components (e.g. lists, buttons, checkboxes, text inputs etc.). However, by providing a pre-configured set, you can expect several advantages:

- All of the components will have been tried and tested, much more thoroughly than you could have done for equivalent components that you 'hand-rolled'
- There are many users who are already familiar with the jQuery Mobile "look and feel", and so will find the applications you used jQM for less intimidating
- There will be a consistency across the framework, so that individual controls will appear to be part of a coherent set, and multi-component user-interfaces will work seamlessly
- Because the developers have been working on jQM for a significant period of time, they will already have tried and rejected many different types of component, saving you the time of going through this learning curve

When you develop a new application using jQuery Mobile, there are several common steps you will need to go through:

1. Create a main HTML file and incorporate the jQM script libraries into it
2. Add 'pages' (html elements) to the HTML file, and add transitions from page to page for some or all of these
3. Add Javascript code that defines the main data structures and application logic for the app. For example, if you were building an app to let a user create notes on their phone, where each note could have a time-stamp, photographs, a list of items (e.g. for a shopping list) etc., then you would have to manage the data for these and provide functions for manipulating that data in Javascript
4. Add code to access an online data source; e.g. so that the application's notes could be uploaded to a web server for sharing with other users, or with the user's other computing systems (desktop machines, laptops, iPads etc.)
5. Package up the application so that it can be used on the user's device even when there is no available internet connection.

In this chapter, we'll go through the first three of these stages. I'll fill a lot of the chapter up explaining some of the detail (and providing links to deeper explanations where these are needed).

Starting a new Mobile Web App

Ask 100 programmers how they would go about building a new program from scratch and you're likely to get 100 different answers. You would expect some overlaps between them (e.g. it is always a good idea to have a very clear description of what you intend to build before you get too far into it), but the various approaches could still be radically different. Approaches that have been popular recently embody such principles as:

- Start with a complete, precise description of what you are building
- Start with a very simple solution and gradually refine it until it is suitable
- Start by defining the major "things" or "entities" that the program will manage (e.g. calendar appointments in a Diary app)
- Start by developing the tests that you expect the completed program to pass
- Start by interviewing the major stakeholders (e.g. end-users, the person paying) to find out their needs and expectations
- Start by (insert favourite dogmatic approach here)…

I've used all of these to some degree (they all have their uses) and have tried to teach new programmers software development using all of them at some point. It took me a while to realize I was missing the main point, which is that while all of these are suitable approaches to building software, none of them are ideal when it comes to *learning to build software*. The difference is that someone who is expected to build software probably knows a lot about what software is made of, what tools can be used, how to test software etc. already. When you're learning, *it is all new*.

Of all of the approaches listed above, probably the most suitable one to apply to learning would be the second in the list: it is always easier to learn to build a simple thing that it is to learn to build what you want. Also, the 'gradual refinement' phases will provide lots of opportunities to explain what is wrong with some aspect of a solution and to introduce new approaches.

Of course, this still leaves us with the problem: how do we define what it is we want to build. General consensus is that this is the hardest question to ask in computing, since there are always very different viewpoints (who will use the software, who will build it, who will pay for it, who has to maintain it once it is in operation). However, this is an issue that we can ignore at this stage. Instead, we can always just start by giving the program a name.

Build a "Shopping List App"

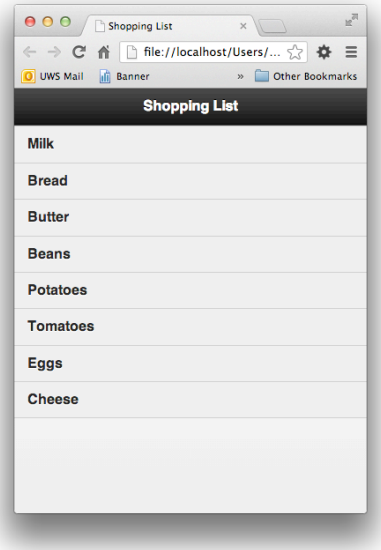
This immediately tells us what we expect the program to do, and also gives us some idea of how it ought to work. For a beginner with a little knowledge of HTML, it might also suggest an immediate solution (warning – it won't be a very good one, but it will make room for discussion of the refinements). Lets start with a straightforward jQuery Mobile list:

```
<!DOCTYPE html>
<html>
<head>
    <title>Shopping List App</title>
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="stylesheet" href=" jquery.mobile.min.css" />
    <script src="jquery-latest.min.js"></script>
    <script src="jquery.mobile.min.js"></script>
</head>
```

```
<body>
    <div data-role="page" id="listPage">

        <div data-role="header">
            <h1>Shopping List</h1>
        </div>

        <div data-role="content">
            <ul id="list" data-role="listview">
                <li>Milk</li>
                <li>Bread</li>
                <li>Butter</li>
                <li>Beans</li>
                <li>Potatoes</li>
                <li>Tomatoes</li>
                <li>Eggs</li>
                <li>Cheese</li>
            </ul>
        </div>
</body>
</html>
```



Listing 5.1: A (trivially) simple Shopping List App in HTML/jQuery mobile code

An initial glance at listing 5.1 and the resulting screen-shot might suggest we're done and can move on to the next job. However, a tiny amount of thought should bring up at least one important question – "What if that's not the groceries I (the user) actually need?". We need to go back to the drawing board, but at least we now have something concrete that we can reason about (which is one of the *real* goals of a gradual refinement approach). I can immediately think of three important questions:

1. How would I add items (i.e. what I actually wanted) to the existing list?
2. How would I remove items from the existing list?
3. Once I have the list I want, how would I indicate on the list that I've just added an item to my shopping basket?

I'm sure there will be more questions, but we can start working on answering these ones straight away.

jQuery Mobile and Lists

As you can see in the highlighted section of listing 5.1, jQuery Mobile provides a simple way to display a list on a device's screen (mostly using plain HTML). Usually we'll put a list in the 'content' `<div>` of a page, and the HTML tags we need are `<ul>` (unordered-list) and `<li>` (list-item). The `<ul>` element contains a whole list, where each item in the list is given as text (or other mark-up) inside a `<li>` element. Also, the `<ul>` element is given a jQuery attribute (data-role) that indicates that it is a "listview" – a special type of list designed specifically as a jQM list for small-screen devices.

Using Javascript (and some jQuery functionality), we can manipulate the items in the list, add new ones and remove existing ones. You should note that the current list's `<ul>` element has been given an `'id'` attribute. That will allow us to create a reference to the whole element using a simple jQuery selector expression:

```
var list = $("#list");
```

Now that we have this, we can insert a new element into the list by simply adding the new item text inside a pair of `<li>..</li>` tags.

```
list.append("<li>Orange Juice</li>");
```

Looking at the screenshot of a version of the page that appears once this code (both lines) have executed, you can see that we're nearly there (but something is not quite right).

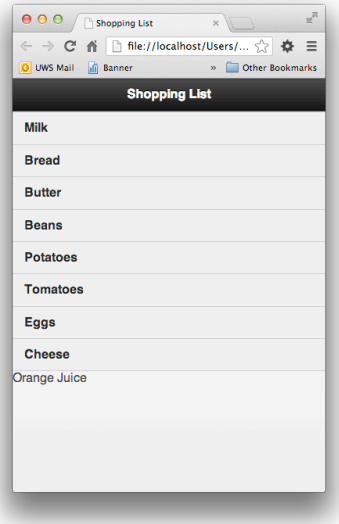


Figure 5.2: The shopping List page with a new item added (note the mismatched item)

The problem with this solution is that the added item looks as if it belongs to a different list – spacing is wrong, it is in a different font and size. A quick Google search using the terms "jQuery Mobile listview add item" brings up loads of hits, most of which provide an answer to this problem: having added a new item to a listview, you then need to 'refresh' the list to make the new items match the style of the list. Adding that code, we now have three lines of Javascript/jQuery code that will append an item to a listview, and it would be sensible to make a function of them:

```
function addNewItem(itemText) {
 var list = $("#list");
 list.append("<li>" + itemText + "</li>");
 list.listview('refresh');
}
```

Listing 5.2: A function to add a new item to a listview

Note that I've generalised the code a bit by adding a parameter to the `addNewItem()` function. The variable `itemText` will refer to whatever string we pass into the function when it is called (e.g. `addNewItem("Orange Juice");`), and we simply wrap that text up in a pair of `<li>..</li>` tags within the call to the list's `.append()` method. This function should be added to a new Javascript file and a `<script>` tag should be added to the HTML file's <head> to link to it. (Note the full listings for this app are provided at the end of this chapter).

Having solved this particular problem (how to add an item to an existing list), we've just created two more; how do we get the user to specify the text that the new item will have?, and how do we execute the `addNewItem()` function to add the item text? (it's as well to accept right now that all of computer programming is like this; the seemingly trivial things we don't even bother thinking about end up being the things that generate the most work. Once you've accepted that, you'll probably begin to understand whether programming is a job that you are suited to or not).

Executing the function is the easy one to answer; recall that jQuery Mobile (actually jQuery) gives us a way to attach a function to a button (or anything that is able to generate an event), so all we need

to do is to call the `addNewItem()` function from an event handler attached to a button on the screen. The button will be our "Add Item" button. What would be nice is if we could, somewhere in the middle of this, ask the user what text (s)he wanted to add to the list.

(A slight divergence) Asking the user something in the middle of a task

In some types of programming, it is very easy to ask the user to provide information. For example, if we were using plain old Javascript in a normal browser, we could write our event handler like this:

```
function Add() {
  var item = window.prompt("What do you want to add?");
  addNewItem(item);
}
```

Listing 5.3: Asking the user for input in a browser

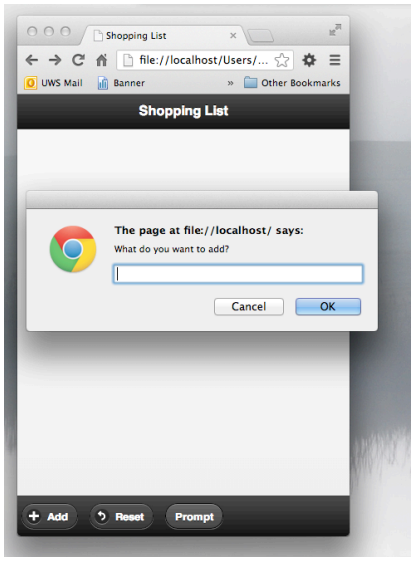


Figure 5.3: The Javascript window.prompt() function - note this is not available in mobile browsers

Unfortunately, we can't do this in a mobile app, since there is no equivalent of the `prompt()` function (or `alert()` or `confirm()`) in mobile browsers. In fact, to do this (seemingly trivial) task, we'll need to provide a whole new page in the HTML mark-up, and incorporate some Javascript event-handling code to operate it. In general programming terms, what we need to build is a dialog box, which is a way of asking the user for information as an aside from the main screen (much like the `prompt()`, `alert()` and `confirm()` features of desktop browsers).

In use, a normal dialog box pops up on the screen in front of the main user-interface, allows the user to enter some information and then is accepted (usually by pressing an OK button) or dismissed (by pressing a Cancel button), at which point the entered information is available to the function call that initiated the dialog box. However, we can't do this in jQuery Mobile because it would break some rules of web pages. Basically, Javascript code is not allowed to ***block*** execution; what this means is that we shouldn't define a function that causes execution to stop while nothing else is happening.

This turns out to be a very common scenario that other programming languages (e.g. Java, C++) allow – stop a program's execution in mid flow to wait for the user to do something. However, Javascript has to work to a higher standard in this respect since otherwise it would be very easy for any website to cause your browser to 'freeze-up'. The `alert()`, `prompt()` and `confirm()` functions

are exceptions in this respect because the browser has full control over them and can make sure that they are displayed in a way that is completely obvious to the user.

The way to get around this problem in mobile Javascript is to use an event to start an operation (e.g. making the dialog box appear), and a second event to complete it (e.g. when the user presses the OK button). This is the standard Javascript way around any situation where there might be a significant and unpredictable amount of time between the start and the end of some operation; typically where the app is waiting for the user to do something or it is downloading Internet data. Instead of the simple two-line function shown in listing 5.3, we will need to create TWO event handlers – one to deal with the user pressing the add button, and one to deal with the user pressing OK (we can ignore the Cancel button because jQM mark-up will handle it for us automatically).

First our dialog box – this turns out to be a simple change to a normal jQuery page- :

```
<div data-role="dialog" data-close-btn="none" id="getItemPage">
    <div data-role="header">
        <h1>Add an Item</h1>
    </div>
    <div data-role="content">
        <form action="#">
            <div data-role="fieldcontain">
                <label for="item">Enter item description:</label>
                <input type="text" id="item"/>
            </div>
            <a href="#" id="ok" data-role="button" data-inline="true"
                                data-rel="back" data-icon="ok">OK</a>
            <a href="#" id="cancel" data-role="button" data-inline="true"
                               data-rel="back" data-icon="cancel">Cancel</a>
        </form>
    </div>
</div>
```

Listing 5.4: The HTML mark-up for a jQuery mobile dialog box

Some specifics about this dialog box:

- It has a `data-role` attribute of "`dialog`" – this affects the way the page appears on the screen
- We've set the `data-close-btn` attribute to "none" – otherwise a close button would be added, and we want to use OK and Cancel buttons to dismiss the dialog
- The dialog/page content is a form, which contains a text box with a label and two <a> buttons (OK and Cancel). The buttons are given a `data-rel` setting of "`back`", meaning they will act as back (close) buttons for the dialog page

Note that this mark up is very specific to the current application – the text input (`id = "item"`) has a label above it to tell the user to enter an item description. If we needed a dialog box to get someone to enter his or her hat size, this one would not do. We'll look into options for making a customizable dialog box later.

We can activate this dialog box in the same way we would move to any page/div in an app – by providing an anchor (<a>) element that links to it. Since the dialog box will be the item that initiates

a major function of the app (adding an item to the list), it would make sense to have put the anchor in the form of a button. Rather than putting this button on the page content area (which is likely to be full of Shopping List), we can place it into a footer area on the page:

```
<div data-role="footer" data-position="fixed">
    <nav>
        <a href="#getItemPage" data-role="button" data-inline="true"
                                data-icon="plus" id="add">Add</a>
  </nav>
</div>
```

Listing 5.5: An anchor-button in the page footer to activate the dialog box

Note that the mark-up for this anchor element is quite detailed – the element's role is to be a button, it is specified as being "`inline`" (meaning that other buttons can be placed beside it), it has an icon which is a "plus" sign and it links (`href`) to our dialog box page from listing 5.4.

Incorporating the Dialog Box

Because we have assigned to the OK and Cancel buttons the attribute `data-rel="back"` in the HTML mark-up, pressing either of these buttons will close the dialog box automatically. However we need to arrange it so that when the OK button is pressed, we create a new item for the list, based on the text in the text-box. The standard jQuery way to do this is to assign an event handler function to handle pressing OK in the `$(document).ready()` code at the top of the Javascript file:

```
$(document).ready(function(){
  // This defines a routine for when OK is pressed...
   $("#ok").click(function(){
        var itemText = $("#item").val() || '';
        addNewItem(itemText);
 });
});
```

Listing 5.6: Attaching code to add a new list item to the OK button

With this code in place, we can get rid of the items that are coded into the HTML page (i.e. Milk, Bread…Cheese), although we need to be careful to leave the empty `<ul>…</ul>` tags behind, since these define the list container. Re-load the app in the browser, and we should have an empty list and a button to let us add items:

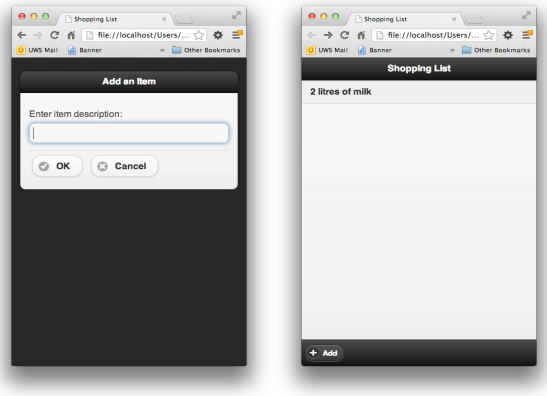


Figure 5.4: Adding an item to the list

De-selecting Items in the list

As you wander round Tesco's with a shopping basket, you will want to mark off items in the list when you've added them to the basket. Alternatively, if this is a To-Do list, you'll need to mark off

items as you complete them. Either way, we need some way of indicating that an item in the list is in a different state – usually we'd think of those items as 'got' or 'completed'.

JQM lists can respond to individual items being tapped (or clicked on with a mouse) provided we handle the events properly. The problem with our application is that we are adding new <li> elements as the program runs, so we're not in a position to say how to handle a click event for the specific items since they are not in the list yet. jQuery gives us a neat way around this problem: we can use a selector to refer to all items with a specific tag type (for example all <li> elements, or even better, all <li> elements inside an element whose ID is "list"). We can attach an event handler to handle a tap (or click) on any of these – even ones that don't exist in the document yet:

```
// This assigns the event to ALL <li> elements in the list
//  – even if they are added later...
$(document).on("click", "#list li", function(){
    $(this).addClass("ui-disabled");
});
```

Listing 5.7: Code to add an event handler to all <li> elements inside the #list

The code in listing 5.7 is a bit special, and shows just how powerful jQuery is for manipulating web pages. First, we are defining an event handler to deal with a whole set of items (all <li> elements inside the <ul> element with the id "list"). Secondly, by using the .on() jQuery method, we are attaching this handler to *all current and future* elements that match that selector.

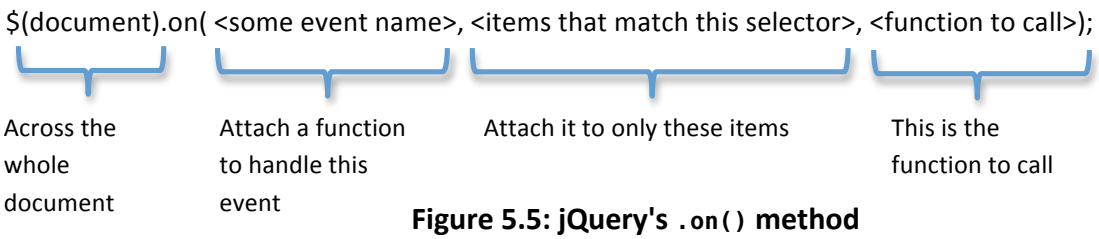


Figure 5.5: jQuery's .on() method

Figure 5.5. shows the general format of the jQuery .on() method, whose purpose is to attach event handler functions to specific events such as the user tapping on an element on the screen (the 'click' event). The final parameter to the .on() method is the function that is to be called. It is standard in Javascript that the identifier 'this' in a function is taken to be the context item – i.e. the item that the function is 'about'. In this case, the use of 'this' is necessary, since the .on() function's selector will refer to ALL of the <li> items in the list – we need to pick out a specific one. 'this' indicates the screen item that was tapped on, and so the code defined *inside the function* in listing 5.7 adds a CSS class to that item.

jQuery Mobile defines a number of CSS (Cascading Style Sheet) classes so that you can manipulate the appearance of elements on the screen. The CSS 'class' "ui-disabled" is used to refer to items on the screen that have specifically been disabled – we can assign it by selecting the element and adding a CSS class to it. Once we've applied the "ui-disabled" class to a specific item in the list, it will be greyed out and it will not respond to events any more.

Note that all of this (adding the event handler and reacting to the event) is done in two lines of code (three if you count the closing brackets).

Now the user can add items to the list and mark items as 'completed'. Since we will also want to be able to remove items (e.g. if we added an item that we later didn't want in the list), we now need to

consider how to do that.

Removing Items

jQuery Mobile provides a Split Button List format so that an additional action can be added to each item in a list. Normally we want to apply one type of action to the left hand side of the list (usually the list text), and a different one to the right hand side (usually an icon/button), with the same pairs of actions for each element in the list. For our purposes, we want the main action to be to de-select the item in the list, marking it as 'disabled' or 'completed'. The secondary action would be to remove the item from the list completely.

Figure 5.5 shows our app with a split-button list. Pressing on the text on the left hand side of an item will disable it; we've already got the code for that, although we will need to make one minor change so we can differentiate between the two sides of an item. We want pressing the "x" icon on the right to cause that item to be deleted.

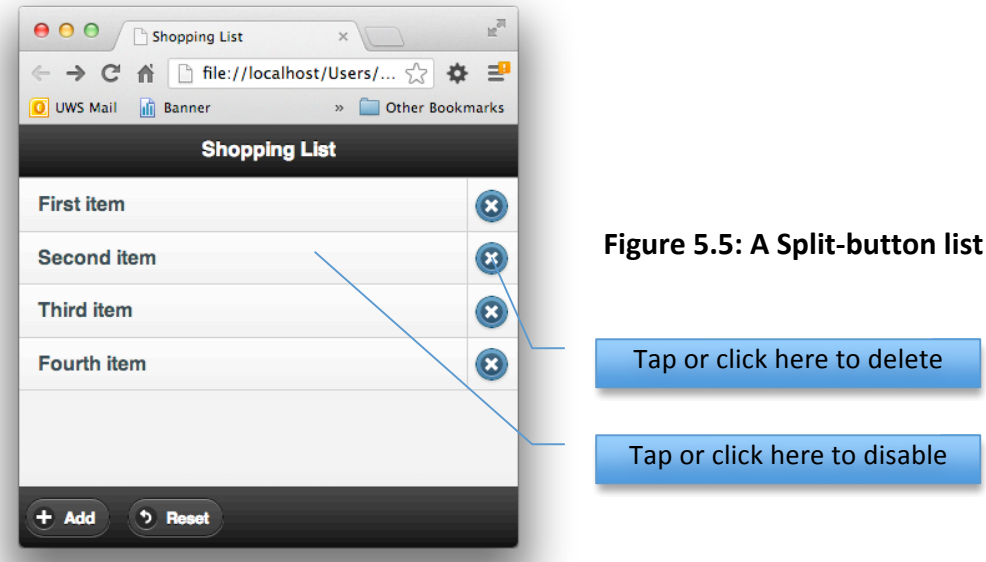


Figure 5.5: A Split-button list

Recall that we add an item to the list in code – see listing 5.2. We need to alter this code to generate a suitable split-button list item. The required format is:

```
<li><a name="disable">Left side text</a>
    <a name="delete" data-icon="delete"
       data-iconpos="notext"></a></li>
```

Listing 5.8: Mark-up for a split-button list item

The mark-up shown in listing 5.8 is a `<li>..</li>` element that contains TWO anchor tags. The first anchor refers to the item in the list (its text) and the second anchor refers to the icon/button at the right. Note that neither element has been given an `href` attribute since that is not needed here (we're not moving to another page). Also note that neither anchor has an id; instead, they have both been given a `name` attribute.

When we wish to select an individual HTML element, we normally use its id attribute in a selector – e.g. `$("#list")`. If we want to select all the elements of a particular *type*, we can use the elements

tag name – e.g. `$("p")`, to select al the <p> elements. However, what if we want to select items in the list, but only those anchor elements on the left (the 'disable' elements), or those anchor elements on the right (the 'delete' elements)?

In this case, we need a selector that narrows the grouping more precisely. `$("a")` would get every anchor element in the document. `$("a[name=disable]")` *filters* that grouping to select only those with the name attribute set to 'disable' – i.e. that would match all of the left hand items in the list. As you can probably guess now, `$("a[name=delete]")` would filter out all of the right hand elements (the 'delete' buttons).

We need to re-write the `addNewItem()` function so that it adds this mark-up to every new item added to the list:

```
function addNewItem(itemText){
    if(itemText){
        var list = $("#list"),
            listitem = "<li><a name='disable'>" + itemText +
                       "</a><a name='delete' data-icon='delete' " +
                        "data-iconpos='notext'>Remove</a></li>";
        list.append(listitem);
        list.listview('refresh');
    }
}
```

Listing 5.9: A revised addNewItem() which marks-up each <li> as a split-button list item

We can now revise the event handler for disabling an item, and add a new one to delete an item:

```
$(document).on("click", "a[name=disable]", function(){
    $(this).addClass("ui-disabled");
});

$(document).on("click", "a[name=delete]", function(){
    $(this).parent().remove();
    $("list").listview('refresh');
});
```

Listing 5.10: Event handlers for the click events applied to either side of the split-list

In listing 5.10, the first handler selects out all anchor elements (<a>) that have a `name` attribute `'disable'`. The second one selects out the anchor elements that are list buttons (where name is set to 'delete'). Note that we're now filtering a selection of ALL of the anchor elements in the document. Since a filter is used, we don't have to worry about selecting only those inside the `<ul>` element.

The second of the two event handlers removes a whole `<li>` element by selecting the `parent()` of the item clicked on; that is the `<li>` element that contains the anchor.

How to lose customers

Our mobile app will now accept new items added to the list, allow items to be marked as 'got'/'completed', and let us clear items off the list. There remains one more issue, which, typically,

is a big cause of frustration for the user. If for some reason the user closes the browser in the middle of creating a list (or closes the browser tab that hosts the shopping list), the next time they go back to the page there will be a fresh list with no items in it. Almost certainly, that user will not use your web-app again.

Until HTML 5, solving this problem involved some quite complex coding using plug-ins, or a web server or various other tricks. HTML 5 includes a specification for "local data storage" that each web page is allowed some of (typically up to 5 Mbytes, or more if you ask politely). This gives us a very simple way to stash a whole or partial list away whenever the user leaves the page (or the browser crashed, or the mobile is shut down). We can use a jQuery event ($(window).unload()) to determine if the browser window is closing and run a short function to store the data. When the app next starts up, we can use the $(document).ready() event handler to reverse this and put the list back in pace.

We would start by defining the functions to be called on these events:

```
function saveList(){
    if(window.localStorage){
        var saveData = $("#list").html();
        localStorage.setItem("shopping_list", saveData);
    }
}

function loadList(){
    if(window.localStorage){
        if(localStorage.getItem("shopping_list") !== null){
            $("#list").html(localStorage.getItem("shopping_list"));
        }
    }
}
```

Listing 5.11: Functions to save (saveList) and re-load(loadList) the contents of the shopping list.

Note that in both cases, we start with a check whether localStorage (the object that manages the process of storing and re-loading data) exists in this browser. All of the current mobile browsers from Apple, Android, Blackberry and Windows Mobile implement the localStorage functions, but it is always safest to check. Provided localStorage is a valid object in the browser, the name will return a truthy value and we can go ahead and use its setItem() function to stash our data (using the syntax: `localStorage.setItem(<item_name, <item_value>);`). Note that the data we are storing is the entire html mark-up inside the `<ul>` element, which is returned by the list element's `.html()` function.

Going in the other direction, we load the data back by first checking that localStorage exists, then checking that data has actually been stored using the specific name ("shopping_list" – not very imaginative but explicit) – the `localStorage.getItem()` function is defined so that it will either return the expected data (always a string) or the value `null`. It is generally a good idea to test if the data exists first; if it does not, we'll avoid assigning `null` to be the contents of the list. The data we stashed can be restored to the list using the list element's `.html()` function again. Note that this is a very useful jQuery function – when we call it like a function (`x = y.html()`), it returns the current html mark-up. When we call it like a subroutine (`y.html(x)`), it assigns the new value. You'll find that many jQuery functions work this way.

Now all we need to do is attach these two functions to the app so that they execute at the right times. We can call `loadList()` at the end of `$(document).ready()`. For the call to `saveList()`, we

need to define a new top-level event handler for the `$(window.unload()` function:

```
$(window).unload(function(){
    saveList();
});
```

Listing 5.12: Saving the shopping list

We now have an app that properly manages a shopping list – we can add new items, mark items off on the list and remove items. We can also shut down the app (closing the browser window or simply navigating away to a new URL) and expect the list to still exist when we next run it. Moreover, the user-interface is exactly what someone used to using mobile applications would expect – buttons and list items are big enough to tap on with a finger, there are subtle animations (e.g. when the dialog box opens and closes) that give the app some sparkle and the colour scheme fits in well with (at least some) mobile platforms.

There are several changes that should be made just to make the app less awkward to use: these would come under the heading of "general tweaks" since they don't change what the app does, but do change the way the user would interact with it (removing several minor annoyances. These would include:

- When the dialog box appears, it should not contain the text left there the last time it was used
- When the dialog box appears, the text cursor should be in it already. Currently, the user has to tap in the text box to put a cursor into it. This means that the on-screen keyboard in a mobile device would not be available until the user tapped the text box
    - We can fix both of these in a single event handler that we would set up in $(document).ready() to handle the event that the dialog box is about to be displayed (the 'pageshow' event)
- It would be useful to have commands to clear the current list and to reset items that have been selected
- It would be nice to be able to keep more than one list going. This opens a bit of a can of worms, since we now need to give each list a distinctive name (not just 'shopping_list'), and we would need to be able to select an existing list for loading. These are bigger issues, but can be coded quite efficiently using jQuery & jQM

The complete code for this app (listed on the next three pages) is tidied up a bit and I've added comments to make it clear what all of the code does. You'll also find it on the module page of Moodle in the Examples and Demos section, and can access an extended version of it that handles multiple lists on your PC or phone using the (shortened) URL - `goo.gl/07eYq.` The code for the extended version is also in the Moodle Examples and Demos section. As we go on to look into Object-Oriented Programming in the next chapter, we'll return to discus some of the other things that jQM can do for our apps.

Shopping.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Shopping List App</title>
    <meta name="viewport"
          content="width=device-width, initial-scale=1" />
    <link rel="stylesheet" href="jquery.mobile.min.css" />
    <script src="jquery-1.9.1.min.js"></script>
    <script src="jquery.mobile-1.3.1.min.js"></script>
    <script src="Shopping.js"></script>
</head>
<body>
    <div data-role="page" id="listPage">
        <div data-role="header">
            <h1>Shopping List</h1>
        </div>
        <div data-role="content">
            <ul id="list" data-role="listview">

            </ul>
        </div>
        <div data-role="footer" data-position="fixed">
            <nav>
                <a href="#getItemDialog" data-role="button" data-inline="true"
                                              data-icon="plus" id="add">Add</a>
                <a href="#" data-role="button" data-inline="true"
                           data-icon="back" id="reset">Reset</a>
                <a href="#save" data-role="button" data-inline="true">Save</a>
                <a href="#load" data-role="button" data-inline="true">Load</a>
            </nav>
        </div>
    </div>
    <div id="getItemDialog" data-role="dialog" data-close-btn="none">
        <div data-role="header">
            <h1>Add an Item</h1>
        </div>
        <div data-role="content">
            <form action="#">
                <div data-role="fieldcontain">
                    <label for="item">Enter item description:</label>
                    <input type="text" id="item"/>
                </div>
                <a href="#" id="ok" data-role="button" data-inline="true"
                           data-rel="back" data-icon="check">OK</a>
                <a href="#" id="cancel" data-role="button" data-inline="true"
                           data-rel="back" data-icon="delete">Cancel</a>
            </form>
        </div>
    </div>
</body>
```

```
</html>
```

shopping.js

```
$(document).ready(function(){
// This handles taps on the OK button in the dialog box...
    $("#ok").click(function(){
        var itemText = $("#item").val() || '';
        addNewItem(itemText);
    });

    // This code puts a cursor into the text-box in the dialog-box when it opens...
    $("#getItemDialog").bind('pageshow', function(){
        $("#item").val('').select().focus();
    });

    // This defines code for a 'reset' button, to re-enable all of the 'selected'
    // list items.  It lets the user undo selections...
    $("#reset").click(function() {
        // Remove the 'disabled' status from all <a> items inside any <li>
        // items inside the element whose id is "list"..
        $("#list li a").removeClass('ui-disabled');
    });

    // This is needed to prevent an Enter key mucking with the user's interactions
    // with the page.  Key-code 13 is the "Enter" key, and pressing this (in a
    // desktop browser) causes a bit of havoc...
    $(document).on('keydown', function(e) {
        if (e.which == 13) {
            e.preventDefault();
        }
    });

    // If a list has previously been saved, this will load it...
    loadList();
});

// Save the current list contents when the browser page closes or goes to a
// different URL...
$(window).unload(function(){
    saveList();
});

// Disable (grey-out) a list item if it is tapped...
$(document).on("click", "a[name=disable]", function(){
    $(this).addClass("ui-disabled");
});

// Delete a list item if it's 'x' button is tapped...
$(document).on("click", "a[name=delete]", function(){
    $(this).parent().remove();
    $("list").listview('refresh');
});
```

```
/**
 * Code to add a new item to the list.  The additional mark-up makes the item
 * a split button list item, with a text side and a button side...
 */
function addNewItem(itemText){
    if(itemText){
        var list = $("#list"),
            listitem = "<li><a name='disable'>" + itemText +
                       "</a><a name='delete' data-icon='delete'"
                       data-iconpos='notext'>Remove</a></li>";
        list.append(listitem);
        list.listview('refresh');
    }
}

/**
 * Function saves the entire list of items to localStorage.
 */
function saveList(){
    if(window.localStorage){
        var saveData = $("#list").html();
        localStorage.setItem("shopping_list", saveData);
    }
}

/**
 * Function asks the user to select a list and then loads
 * the list into the app.
 */
function loadList(){
    if(window.localStorage){
        if(localStorage.getItem("shopping_list") !== null){
            $("#list").html(localStorage.getItem("shopping_list"));
        }
    }
}
```