Lab 4: Mobile Event Handling

Objectives of this lab session:

- *Make use of mobile-specific user-interface events in apps*
- *Add library functions to a web app, promoting a modular structure to the app*
- *Adding Google Maps functionality to the app*
- *Setting up the app so that it can be properly "installed" on a device*

Resources required:

- *A current web browser (ideally Google Chrome)*
- *WebStorm IDE*
- *jQuery & jQuery Mobile libraries (a link to these is fine unless you wish to work offline)*
- *Files from the Moodle page for this lab*

Part 1: Create a Simple Multi-Page app

We will start with a simple jQuery Mobile app that has several pages…

1. Start a new project (Lab 4) and add a HTML file to it (**lab4.html** is a suitable name).
2. Link in the jQuery and jQuery Mobile libraries and add the **<meta>** viewport element as usual (see labs 1 & 2). Add an another <script> link to bring in the Google Maps code:
   ```
   <script type="text/javascript"
   src="http://maps.googleapis.com/maps/api/js?sensor=true"></script>
   ```
3. Find the files **maps.js**, **popups.js** and **utils.js** in Moodle and add these to the project (drag-and-drop files from an Explorer/Finder window directly on to the Project window in WebStorm). Add links to these files at the head section of the HTML file – the links must come *after* the links to jQuery & jQuery Mobile scripts
4. Add a new JS file to the project (call it **lab4.js**) and add a link to it, after the other JS file links
5. Add a jQM page-div with the usual header, content and footer element-divs. Give this page the ID **"homepage"**
6. The content div of the home page will be a listview with (currently) no list items in it:
   ```
   <ul data-role="listview" data-inset="true">
   </ul>
   ```
7. Add an extra <div> to the page *after the page footer*, with the following mark-up
   ```
   <div id="panel" data-role="panel">
     <div><h2 id="time"></h2></div>
     <a href="#" id="close" data-role="button" data-rel="back">Close</a>
   </div>
   ```
8. This mark-up adds a Panel element to the page that it is placed in. The panel can be opened by a normal **<a>** link
9. Add three more pages to the app, with the IDs **"listpage"**, **"mappage"** and **"canvaspage"**. Give each page an identifying header (e.g. **"List Page"** inside the header div). Add a listview to the **"homepage"** content section with four **<li>** elements in it (**<ul><li>…</li><li>…** etc.), and add **<a>** links inside each of the **<li>** elements; link these to the panel and the three other pages in the mark-up

You should now be able to ***test the app***, and tap on the links to move to the other pages and the panel. Note that all of the pages should have back buttons in their header (put there automatically if you've used the WebStorm live templates created for the module to create the mark-up). You should now spend a little time tidying up the app – for example:

- Remove the **"back"** button form the header of the home page
- Add some header and footer text to each page so that they can be easily identified
- Add an **"About…"** button (an **<a>** element with **data-role="button"**) to the home page and give this the id **"about"**.

Test the app at each stage to make sure that none of your tweaking introduces errors.

Part 2: Add an About… page

The **popups.js** library file you added to the project contains definitions for a number of dialog-style boxes. These add functionality similar to the **alert()**, **confirm()** and **prompt()** functions available in a web-browser app, which, although they are standard Javascript web-browser functions, are disabled by jQuery Mobile. The four dialog types added are:

- **popupAlert(<title>, <message>);** – which displays a message in a box with a close button
- **popupConfirm(<title>, <question>, <function for Yes answer>, <function for No answer>);** - which allows you to ask a simple confirm-style question
- **popupPrompt(<title>, <prompt-message>, <function for OK>, <function for Cancel>);** - which lets you ask the user to enter some data
- **popupLogin(<function for OK>, <function for Cancel>);** - which displays a login dialog box so the user can enter a user-name and password

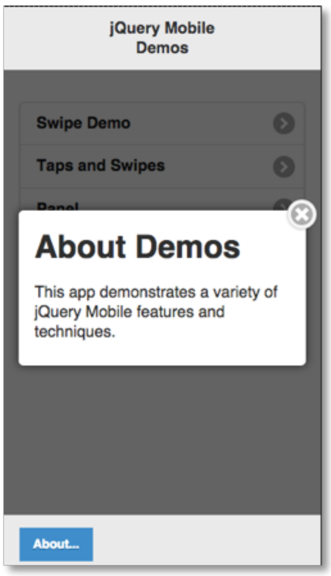
You added an **"About…"** button to the **nav-bar** section of the home page, and we will use one of these functions (**popupAlert()**) to make that work. For this, you will need to add a new JS script file to the project (call it **lab4.js**) and link this to the HTML file. To complete the About box, add a **$(document).ready()** function to the new Javascript file, as follows:

```
$(document).on("pageinit", "#homepage", function() {

  $("#about").on("click", function () {
    popupAlert("About Demos", "This app demonstrates a variety of\ " +
          "jQuery Mobile features and techniques.");
  });

  // More code here...

});
```


jQuery Mobile
Demos
Swipe Demo
Taps and Swipes
Panel
About Demos
This app demonstrates a variety of jQuery Mobile features and techniques.
About...

We can now **test the app again** – open the app in a browser and press the "About…" button on the home page. You should see the About dialog box, shown on the right. If you don't check your code for errors (remember that WebStorm puts a red mark in the margin if it detects an error.

Note that an alert box is the easiest option since there is no data to retrieve from the dialog. We'll see in the next exercise how to get information back from a dialog box.

Part 3: Activating the Panel

The Panel element added to the "home" page is normally used as an off-screen menu or a list of configuration items. In this app, we'll simply use it to display the time. Of course it would be far more efficient to display the time in the header of the home page, but this example will illustrate how to use a panel.

Displaying the time in an app is easy enough; simply set up a timed call to a function that displays the time – **setInterval()** is the best function for this purpose:

1. Add a call to **setInterval()** in the **$(document).on("pageinit", …)** function (at the **"More code here…"** comment):

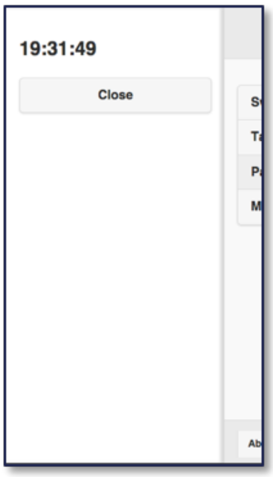
```
setInterval(function() {
  $("#time").text(getTime());
}, 1000);
```

   Note that the function to be called repeatedly calls on the **getTime()** function defined in **utils.js**. Once per second this updates the text content of the heading inside the panel's content **<div>**.

2. Add code to close the panel on a button press to the end of the **$(document).on("pageinit",…)** event handler (immediately after the code added in step 2:

```
$("#close").on('click', function() {
  $("#panel").panel("close");
});
```

   Note that there is no real need for this code, since clicking on the screen off the panel will close it. However, users are often put-off by the lack of an *obvious* way to get out of a part of a program, so it is usually worthwhile adding this functionality and the button it targets.


19:31:49

Again, it is worth testing the app – when you click on the "Panel" list item on the home page, you should see a display like that shown on the right.

Part 4: Manipulating a list with Touch events

jQuery Mobile includes support for the interaction events specific to mobile phones and tablets – i.e. it assumes a touch screen and provides "tap", "swipe" and other events to support this. To investigate the way that touch events work, we will add a page to manipulate a listview:

1. Go to the **"list"** page of the app and give it the heading **"Swipe Demo"** (obviously you can call it whatever you like, but this name is descriptive). While you are in the header section, you could also remove the **"Back"** text from the back button, and add the property: **data-iconpos="notext"** – both of these are necessary to get a back button that looks good without a need for the descriptive text

2. Add a **<ul>** element to the page content area (**jqmlist + Tab** is a useful shortcut); make sure the element has the properties **data-role="listview"**, **data-inset="true"** and **id="list"**

3. Now add a number of **<li>** elements inside the **<ul>** - a useful shortcut for this is to enter **li*6** (if, for example, you wanted 6 **<li>** elements). Add some text to each of them – month names, members of your favourite football team, names of the dwarves from the Snow White or the Hobbit – whatever

4. Go into the "footer" area of the page and, inside the **navbar** section, add an **<a>** link with the following mark-up:

```
<a href="#" id="add">Add...</a>
```

This completes the HTML mark-up for the "list" page.

Dealing with Events efficiently

To make the list page work, it will need some Javascript/jQuery code to define event handlers, similar to the **$(document).on("pageinit"…)**. We could fill lab4.js with a number of **$(document).on(…)** function calls to set up event handlers. However, each call to the selector **$(document)** has a processing overhead and so it would be useful to minimize the number of times this selector is used. The jQuery developers have thought of that and implemented the **$()** selector functions so that a single selector can be used many times. The general principle is that jQuery code is written so that you can string function calls together, like:

```
$(<selector>).doThis().doThat().doAnotherThing();
```

Using this principle, we can organise the main application event handlers into a block, rather like a case or switch statement in general programming languages.

1. Add a new event handler to the app, immediately after the closing bracket of the first **$(document).on("pageinit"…)** call.

```
$(document).on("pageinit", "#homepage", function() { // existing code

    // Code to initialize the app (already exists).

}).on("pageinit", "#listpage", function() {          // A new event handler

    $("#add").on("click", function (evt) {
            var item;
            popupPrompt("Add Item", "Enter list-item text:", function() {
           item = getPromptValue();
           $("#list").append("<li>" + item + "</li>").listview("refresh");
        });
    });

   // The code to delete a list item will go here.

});
```

Note the use of the **poupPrompt()** function. This takes a heading, a prompt to the user and a function to be called if the user clicks **OK** in the dialog box. You can add a second function if you want to perform some action when the user clicks on **Cancel**. Inside the function, a call to **getPromptValue()** will return whatever the user typed into the dialog box. If you now test the app, you will find that you should be able to add new items to the list. We now need to add code to allow items to be *removed* from the list.

2. Add the following code to the listpage's "pageinit" event handler (note – at the comment "*The code to delete a list item will go here*"):

```
$("#list").on("swiperight", "li", function () {
   var li = $(this);
   li.animate({"margin-left": '+=' + $(window).width( )}, 400, function() {
      li.remove();
   });
});
```

You should carefully examine the way this event handler is constructed. First, the initial selector is **$("#list")**, which means that it will apply to the "list" **<ul>** element in the app, and according to the event name, will apply to this element when the user swipes a finger or mouse with a button pressed to the right on the list. However, the event target is modified by the next parameter (**"li"**), which is called the *context* element. When the event happens, the handler will select the item in the list that the swipe was made on. Inside the handler function, **$(this)** refers to that **<li>** item.

The **li.animate()** function will therefore act on whichever **<li>** element was swiped, causing its left hand margin to be incremented by the width of the screen (**$(window).width()**). This would, on its own, cause the list to display weirdly, since one of its elements would be off to the side of the screen. However, after the animation finishes (in 400 mS), the offending item is removed. Note this – the **$("#list").on("swiperight", …)** event attaches an anonymous function to the event, which itself attached an anonymous function to the **animate()** function call (the last function is executed once the animation has completed) – this deep nesting of anonymous functions is common in jQuery applications.

You should now test the app again. Move to the List page and try swiping some elements to the right – they should animate off the screen, leaving the list one **<li>** item shorter. Note that this will also be the effect if you swipe on **<li>** elements you add to the list.

Part 5: Using Google Maps

Google maps works as well in jQuery Mobile as in any other browser app. However, because jQuery Mobile pages are not rendered until just before they are actually displayed, the usual mechanism of initializing a <div> element to fit a map does not behave in quite the same way. It works out best to handle drawing a map in two stages – first re-size the <div> element the map will be displayed in, which can be done in the background, and secondly draw the map of the re-sized <div> element. Since we can also use geo-location to decide which area the map should depict, these two jobs can be neatly organised in two top-level event handlers.

1. Add "header" and "content" divs inside the **<div data-role="page" id="mappage">…<div>** element. There is no need for a "footer" section, but you can add one if you like. Inside the 'content' div, add a new div element - **<div id="map-canvas">**. This will be the place where the map is drawn.
2. Add these directly after the list-page's "pageinit" event handler (the arrow below is to indicate the closing bracket of the **.on("pageinit"…)**

```
← .on("pageshow", "#mappage", function() {
   // Can't draw the map until the page is visible, since g-maps is a bit incapable
   // of a re-size...
   if(navigator.geolocation) {
      navigator.geolocation.getCurrentPosition(function (pos) {
         drawMap(new google.maps.LatLng(pos.coords.latitude,
                                        pos.coords.longitude), "map-canvas");
      });
   }
}).ready(function() {
   // Re-size the map <div> as soon as the document is loaded...
   google.maps.event.addDomListener(window, 'load', function() {
      initializeMap("map-canvas");
   });
});
```

3. Test the app. When you move to the map page, you should find that the map element will fill the content area (with a small margin around it)

End of Lab 4