HTML5 & Javascript: Lab 7

Objectives of this lab session:

- Use sessionStorage and localStorage to pass data between web-app pages
- Create a multi-page application that has access to the same code and in-memory data

Resources required:

- A current web browser (ideally Google Chrome)
- WebStorm IDE

Part 1: Using sessionStorage

In this part of the lab you'll create a very simple web app that passes data between two HTML documents.

Basic Principles

HTML5 incorporates a sessionStorage API. This works like localStorage, but stores data only during a single web session – i.e. until the user navigates the browser page to a site that is not part of the app.

sessionStorage has the same API as localStorage:

- sessionStorage["key"] = data; // Stores data under the specified key
- sessionStorage.key = data; // As above, but using dot notation to access a property
- data = sessionStorage["key"]; // Retrieves data if the key exists
- data = sessionStorage.key; // As above.
- sessionStorage.removeItem("key"); // Removes the specified item
- sessionStorage.clear(); // Removes all items.

A simple Example

This will be a trivial app that simply demonstrates the principle:

1. Create a new WebStorm Project with the name Multi-page
2. Add two html files to it – page1.html and page2.html
3. Add a single Javascript file – page-data.js

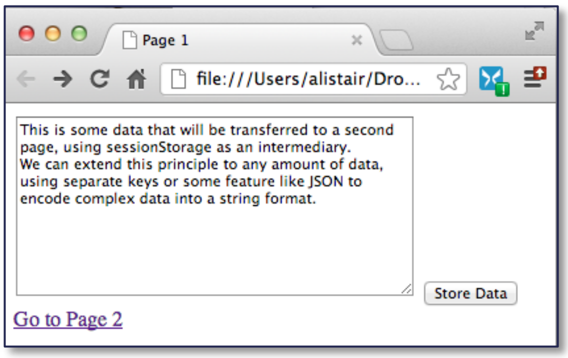
The first page will be one where we create some data. The easiest way to do this is to have a form element – e.g. a textarea control – on the page.

```
<!DOCTYPE html>
<html>
<head>
    <title>Page 1</title>
    <script type="text/javascript" src="page-data.js"></script>
</head>
<body>
    <textarea id="input" rows="10" cols="40"></textarea>
    <button id="save">Store Data</button>
    <a href="Page2.html">Go to Page 2</a>
</body>
</html>
```

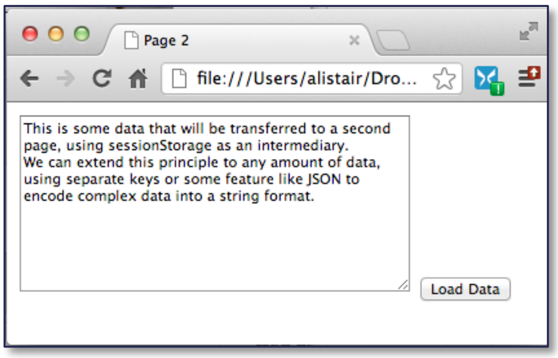
The second page will be very similar:

```
<!DOCTYPE html>
<html>
<head>
    <title>Page 2</title>
    <script type="text/javascript" src="page-data.js"></script>
</head>
<body>
    <textarea id="output" rows="10" cols="40"></textarea>
    <button id="load">Load Data</button>
</body>
</html>
```

These two pages simply give us enough controls to enter some data on one page, save it, move to the second page and re-load the data.



This is some data that will be transferred to a second page, using sessionStorage as an intermediary.
We can extend this principle to any amount of data, using separate keys or some feature like JSON to encode complex data into a string format.



This is some data that will be transferred to a second page, using sessionStorage as an intermediary.
We can extend this principle to any amount of data, using separate keys or some feature like JSON to encode complex data into a string format.

All that we now need is the code to make this work:

```
var input, output, save, load;

function storeData() {
    input = document.getElementById("input");
    sessionStorage.pageData = input.value;
}

function loadData() {
    document.getElementById("output").value = sessionStorage.pageData;
}

window.onload = function () {
    if( document.getElementById("save")) {   // Is this the save page?
        save = document.getElementById("save");
        save.onclick = storeData;
    }
    if( document.getElementById("load")) {    // Is this the load page?
        load = document.getElementById("load");
        load.onclick = loadData;
    }
};
```

The only notable feature of this code is that because the "load" and "save" elements have different names, we've had to check for the existence of an element before trying to use it.

Revising the Appointments app

In this part of the lab, we'll revise the Appointments application so that the input/edit form is shown on one page and the list of appointments is on a different page. Note that since we're already storing appointment data in localStorage, we won't have to employ sessionStorage for this. However, sessionStorage will be useful for passing status variables between pages – for example to indicate the 'current' Appointment.

Since the appointments app has undergone a number of changes over the earlier labs, it is better is we start from a common version. Lab7.zip, in Moodle Week 9, contains an updated version of the Appointments app, which the following changes:

- There are now two html pages and three Javascript files (plus the jQuery library). This effectively splits the project into three parts
    i. The Appointment type, which is in the file appointmentType.js. This contains the code for the Appointment type and the collection of appointments, including functions to save and load the collection to/from localStorage
    ii. appointmentList.js contains the code for displaying the HTML table of appointments, and includes code for interacting with the appointments table view, and code to generate a Google map of the postcode in an appointment. AppointmentsList.html is the web page that works with this Javascript file
    iii. appointmentForm.js has the code necessary for creating an Appointment from the HTML form controls in AppointmentsForm.html, and displaying an Appointment on these
- The combination of files as it is just now acts like two separate HTML apps that just happen to use the same data store. As a result, if you open AppointmentsForm.html, add details and press OK to create an Appointment, and then open AppointmentsList.html, you will get a view of all Appointments including the one just created
- The two HTML pages do not interact, except through the data stored in localStorage. It is the job of this part of the lab to fix that.

The Revised User Interface

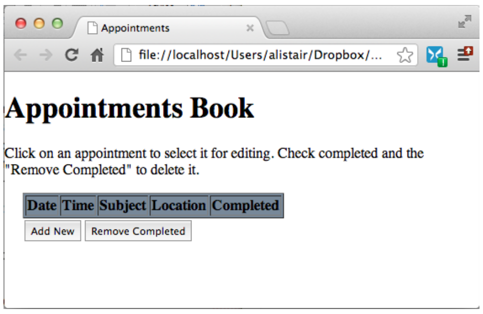
Download Lab7.zip from Moodle week 9, and unzip the archive into a working folder. Note that there are two HTML files – AppointmentsList.html and AppointmentsList.html, and three Javascript files (in the KS) sub-folder – appointmentType.js, appointmentList.js and appointmentForm.js. These contain the separated out components of the previous app. There is one common Javascript file (appointmentType.js) and one file for each HTML page.

To complete the work needed to make the two HTML pages interact, we will need to do the following:

- Arrange it so that when the user clicks on a row in the HTML table on the AppointmentsList page, the other HTML page will open displaying the details of that appointment
- Arrange it so that when the used clicks on the Add New button in the AppointmentsList page, the AppointmentsForm page will open ready to enter a new appointment
- Add code to the AppointmentsForm page (in appointForm.js) so that when the user presses the OK button, the details on the form are stored and the application switches back to the list view. There is an additional problem with this, since the page will need to know whether the user was editing an existing Appointment or adding a new one
- Add code so that when the user presses the Cancel button in AppointmentsForm, the application switches back to the list view with no change

Interacting with the List View

1. Open the AppointmentsList.html file in a browser. You should see a HTML table with no entries (just a header row) and a couple of buttons, like:



Appointments Book

Click on an appointment to select it for editing. Check completed and the "Remove Completed" to delete it.

Date	Time	Subject	Location	Completed

2. Now open the appointmentList.js file in the WebStorm editor (in the JS folder). Go to the end of the file and examine the $(document).ready() function. This loads the appointment list (both apps will do that) and sets up the two buttons on the form with event handlers – addNew and removeCompletedAppointments. The second of these is fully coded already (with some additional comments describing an alternative way to do the job). addNew contains a call to a selectAppointment() function, passing the value -1 to it. The purpose of selectAppointment will be to open the AppointmentsForm page with a specific Appointment in place. However, a call to it with the value -1 will be a signal to create a new Appointment (since appointments will be numbered anything from 0 up, -1 is a safe choice for meaning “no appointment”)

3. Add some code to the selectAppointment() function (around line 84):

```
function selectAppointment(index) {
    sessionStorage.current = index;
    // Open the edit window...
    open("AppointmentsForm.html");
    // ...and close this one.
    window.close();
}
```

   This code stores a value (the -1 passed to the function in this case) in sessionStorage under the name “current”. It then opens the second HTML page and closes itself. We can expect the other page to open ready to accept new appointment details

4. Open the appointmentForm.js file and examine the code in $(document).ready(). This again loads the appointments list, then sets up variables to refer to the fields on the form. It then fills up the time <select> with a list of times of day. Finally it attaches event handlers to the OK and Cancel buttons. If you fill in some appointment details and press the OK button, a new appointment will be created and the list saved (examine the addOrUpdate() function just above ready() – currently this just adds a new Appointment).

5. We need the second form to close and the first to re-open when ether the OK or Cancel button is pressed. Add the following code to the end of addOrUpdate():

```
function addOrUpdate() {
     addNew();    // Existing code
     saveList();  // Existing code.
     // Add these two statements…
     open("AppointmentsList.html");
     window.close();
}
```

   If you return to the AppointmentsList page, click on Add New, fill in details and then click on OK, you should find that you return to AppointmentsList with the new Appointment visible.

6. We need to do the same for the cancel button, although of course in this case you will not want to add a new appointment or save the list – just open the List View and close the current window. Add code for this to the cancel() function.

To recap – we've now altered these two separate web pages so that by pressing the New Appointment button on the list page, we move to the form page where data can be entered. When we complete the form and press OK or Cancel, we return to the list page with the appointment either added or not (depending on the button pressed). Our next goal needs to be to edit an existing appointment.

When we switch from the list page to the form page, we need to indicate which appointment, if any, is to be edited:

7. In the appointmentList.js file, note that there is a selectApppointment() function in which we stash the index of the an appointment in sessionStorage. How this number gets passed to selectAppointment is interesting:
   a. In the appointmentType.js file, look at the Appointment.prototype.tableRow() method. In previous versions of the Appointments app, this was getting quite messy and difficult to follow, so some "tidy-up" work has been done. You should be able to see that the table row is now made up by calling a generic buildTableRow() function (not associated with the Appointment), passing a main function name ("selectAppointment"), a row index and an array of tableData() function calls.
   b. The tableData() function simply wraps <td>..</td> around each element. Most of the elements are simple method calls for the Appointment (getDate(), getTime()…), but two are a bit more complex…
   c. btnText has been set-up earlier in the Appointment.tableRow() method – it takes the postcode, the index of the appointment in the list and a function to call ("showPos"), and builds a string:
      ```
      <button id='" + index + "' onclick='showPos(" + index + ")'>PA1 2BE</button>"
      ```
      This defines a button that, when clicked, will call showPos, passing the index of this appointment into it (with the appointment's own postcode).
   d. checkBox() does a similar job to generate the HTML for a check box, including passing the index of the appointment to the function included (updateStatus, in this case – this toggles the appointment's completed property value).

   Note that when Appointment.tableRow() is actually called (from the updateList() function in appointmentList.js), the function is called with the appointment's array index. Putting all of this together, each table row will include three clickable elements:

- the table row itself, so that if we click anywhere on it, a call to selectAppointment() will be made, including the appointment's index
- the postcode button, so that a click on this will call showPos()
- the completed checkbox, which will change the appointment's completed setting to the opposite of what it previously was
- Note that both the showPos() and the updateStatus() functions (in appointmentsList.js) end with a call to event.stopPropagation(). Without this, the same click event would be passed on to the tableRow's event handler.

8. To work on the selected appointment, we need to add code to the ready() function in appointmentForm.js to select whichever appointment is indicated by the sessionStorage.current value. Add a statement to the end of $(document).ready():

   ```
   showAppointment(sessionStorage.current);
   ```

   The consequence of this will be that the form will open showing the details of an existing appointment. The user can edit this.

9. Now amend the code in addOrUpdate():

```
function addOrUpdate() {
   var current = sessionStorage.current;
   if(current > -1) {
      updateExisting(current);
   } else {
      addNew();
   }
   saveList();
   open("AppointmentsList.html");
   window.close();
}
```

The addOrUpdate() method now clearly has two possible outcomes – either adding a new appointment from the data in the form fields, or updating an existing one based on this data.

User-Interface Strategies

The objective of this lab has been to demonstrate how, by passing simple information around in sessionStorage, we can make different HTML pages act as components of an application. This is a good way to work, since the effect is to simplify each page, compared to placing all of the edit/update/view/list code into the same HTML and JS files. Adding new pages with associated code should not increase the complexity of the app.

However there are other approaches to splitting an application up into separate user-interfaces. One simple one to manage is to put all of the UI parts into the same HTML page (and all of the JS code into one JS file), but put each separate user-interface (forms, tables, lists etc.) into a separate <div> element. We can then show and hide the different user interfaces using simple jQuery (or plain Javascript) code, e.g.:

```
$(“#switching-element”).click( function() {
    $(“#div-to-close”).hide();
    $(“#div-to-open”).show();
};
```

where “#switching-element” is the id of whatever control causes a switch of user-interface elements (e.g. the “OK” or “Cancel” buttons on the Appointments form), “#div-to-close” is the id of user-interface you wish to hide, and “#div-to-open” is the id of the one you wish to show(). If you want to get really fancy, you should look into jQuery effects for animating HTML elements (http://api.jquery.com/category/effects/). Have a look at http://viralpatel.net/blogs/how-to-apply-html-user-interface-effects-using-jquery/ for a tutorial.

Rebecca Murphey does an excellent inline tutorial for jQuery at http://jqfundamentals.com/. Now go and add some of this stuff to your projects.

End of Lab 7