HTML5 & Javascript: Lab 4

Objectives of this lab session:

- Get practice using new HTML5 features
- Apply localStorage and mapping in a practical application

Resources required:

- A current web browser (ideally Google Chrome)
- WebStorm IDE
- A completed Lab 3, containing the full user-interface for the Appointment type.

Part 1: HTML 5 feature demos

In this part of the lab, you will get a chance to implement HTM5 features:

- using the audio tags
- using simple geo-location
- using simple localStorage

In the second part of the lab, you will add some of these features to the existing Appointments application built in the previous two labs.

Audio & Video tags

In HTML 5, audio & video is now considered to be a core part of the HTML standard where previously add-ins (such as Flash player) were required. In this part of the lab, we'll add a simple audio component to a page. You can try the same with a video component for yourself, since the mark-up principles are the same (see Dive into HTML5 – http://diveintohtml5.info/).

1. Start a new project (Lab4) and add a HTML file to it (audio.html is a suitable name).
2. Add an appropriate <heading> tag. The one I used was:

```
<heading>
  <h1>HTML5 Audio</h1>
  <p>Audio files downloaded legally from <a href="http://www.epitonic.com">Epitonic</a></p>
</heading>
```

3. You will need to have at least one audio file to make this part of the lab work. You can choose any format from MP3, WAV, WMA for a Windows machine, MP3, AIFF or OGG for a Mac, or MP3 or OGG for a Linux machine. If found the Epitonic site a good source of audio files which are DRM-free and free to download legally. For now all you need is one (legal) audio file in a suitable format. Place the file in a <media> sub-folder of the project folder.
4. Add an <audio> tag below the heading as follows:

```
<audio id="audio" src="{your audio file here}" controls="controls" preload="auto">
</audio>
```

   Obviously, you should replace the {your audio file here} comment with the name and path of the audio file you downloaded – e.g. "media/my_mp3_file.mp3".

5. Open the web page and check that your audio file plays properly, and that you can use the controls to pause, skip etc.

Controlling what gets played

For some applications, embedding an audio file as we have done is enough. However, if you wanted to create a site that allowed the user to choose which file to play, we'll need to add some script.

6. Download another two or three audio files to your machine. We will build some User-Interface to allow the user to choose from these files.
7. Add a <select> tag to the HTML file – place it between the page header and the <audio> tag. Add some options (one for each audio file) to the select group, coded as follows:

   ```
   <select id="track">
     <option>none</option>
     <option value="{your track 1 file here}" >Track 1 Title</option>
     <option value="{your track 2 file here}" >Track 2 Title</option>
     <option … etc.>
   </select><br/>
   ```

   Obviously you should change the option values and Track titles to match whatever files you have included in the app, remembering to specify the track paths properly (media/…).
8. Add a Javascript file (e.g. audio.js) to the project and add a link to it at the head of the HTML file. At the top of the file, declare two variables – select and audio.
9. We will need to provide two functions within the Javascript. The first of these will create Javascript objects for the <select> and <audio> tags in the HTML document - this is best done in a function that is fired by the window.onload event

   ```
   function initialize(){
     select = document.getElementById(("track");
     audio = document.getElementById("audio");
   }
   ```

   Add this code to the Javascript file and, at the bottom of the file, add a statement to attach it to the window.onload event (see previous labsheets if you've forgotten).
10. The second function should react to the user selecting an audio file from the list and play this file. Suitable code for this is:

    ```
    function playSelectedAudio(){
      var track = select.value;
      audio.src = track;
      audio.load();
    }
    ```

    Add code to the initialize() function to attach this code to the select.onchange event.
11. Re-open or refresh the page in the browser and select a track to play. If it does not play, have a look at the source code in the chrome developer tools, where there is likely to be an error mark.

Geo-location

HTML5 introduces a geo-location API so that a web page can determine the location of the user. The feature is very flexible and can use the best available from a number of possible location information – the IP address of the machine the browser is running on, triangulation from phone towers or a GPS chip on the device.

To create an app that simply shows the current location of the browser:

1. Add a new HTML page to the project – call it location.html.
2. Add a new Javascript file – location.js. Add a link to it in the <head> section of the HTML file.
3. Place a <div> in the body of the HTML file with an id…

   ```
   <div id="location"></div>
   ```

   We'll place text describing the current location into this div.
4. Add the following code to the Javascript file…

   ```
   function showLatLng(position){
     var location = "Lat: " + position.coords.latitude +
                    "Long: " + position.coords.longitude;
     document.getElementById("location").textContent = location;
   }
   ```

```
window .onload = function(){
    if(navigator.geolocation){
        navigator.geolocation.getCurrentPosition(showLatLng);
    } else {
        alert("Location data is unavailable!");
    }
}
```

5. Open the HTML file in a browser – however, DO NOT USE CHROME for this. Chrome is configured to not allow geo-location data (which it considers to be a privacy issue) in a browser run from a local file. It will allow location data when run through a web server, but for now we won't be doing that. Firefox does allow location data to be accessed by local files, and by default enables the feature. As soon as the page opens, the message showing latitude and longitude should pop up.

This is a very trivial geo-location application – a more normal one would display a map showing the current location or some other location and a route between here and there. We'll add a bit of mapping to the appointments application as an option later in this lab.

Local Storage

HTML5 provides a way to store app data (i.e. data that is created by using a website) so that it can be retrieved when the page is revisited. Although browsers have always been able to store some local data using cookies, these have always had a security implication (cookies are written and read from server-based code, so are more about providing information *about* the client to the server). HTML5 local storage is client-side, so data can only be written and read from code within the browser.

To make a very simple app for storing data:

1. Add a new HTML page to the project and call it storage.html
2. Add a Javascript page (storage.js), and add a link to it in the head of the HTML document
3. Lay out some user-interface controls in the body of the HTML page as follows:

```
<body>
    <h1>Local Storage</h1>
    <textarea rows="10" cols="50" id="text"></textarea>
    <br/><button id="save">Save</button>
    <button id="load">Load</button>
</body>
```

4. In the Javascript file, add variables to reference the textarea and the two buttons on the page (text, save and load), and assign a function to the window.onload event so that when the page opens these three variables have the various controls assigned to them
5. Add two functions to the Javascript file as follows:

```
function doSave(){
    var txt = text.value;
    localStorage.storedText = txt;
}

function doLoad(){
    text.value = localStorage.storedText;
}
```

6. Now add statements to window.onload's function to assign the doSave() function to the save button, and the doLoad() function to the load button.
7. Open the page in the browser (preferably Chrome, although any would work) and enter some text into the textarea control. Then press the save button and then browse away from the page (or close down the browser). Finally, re-open the page and press the load button – the text you entered earlier should be retrieved and put into the textarea
8. While the page is open, open the Chrome Developer tools and click on the Resources tab at the top. Go to the Local Storage section (on the left of the developer tools window and (if needed) click on the black arrow-head to open it up. On the right of the tools area you should see an entry for

storedText (a name that was provided by the Javascript code), and whatever you entered into the textarea should be shown next to this. If you wanted to, you can right-click on this entry and delete it.

Part 2: Adding persistence to the Appointments app

Now that we've had a look at some of the more useful HTML5 enhancements available (there are more that we will make use of in later labs), it would be a good idea to apply some of them to the app we've been building over the past few weeks. As a minimum, a user has the right to expect an application to keep track of data a user has entered, so we'll start by giving the Appointments app the capability of storing the appointment data.

To continue you will need a completed copy of Lab 3. If you finished it last week, you can continue with that code. Otherwise, you'll find a completed version of Lab 3 on Moodle in the Week 5 folder.

1. If you want to keep all of the labs separate, copy the files from Lab 3 into a new folder (e.g. Lab4). Open this lab and remind yourself of the various features and code within it
2. Run the lab – you should find that an empty table of appointments is displayed, indicating that the list of appointments has no entries in it just now. We will write code to store the list of appointments to local storage every time a new one is added
3. Add a new function to the appointments.js script file:

```
function saveAppointmentList(){
    var appts = JSON.stringify(appointments);
    if(appts !== ""){
        localStorage.appointments = appts;
    } else {
        alert("Could not save appointments at this time");
    }
};
```

4. We need to arrange that this function is called every time a new appointment is added in the app (also every time one is deleted or altered, but I'll leave that as an exercise for you later). The ideal place for this is the showTable() function which you added to the app in Lab 2. Insert a call to saveAppointmentList() at the end of the showTable() function definition.

What has happened here is this. The function JSON.stringify() is a part of the JSON (JavaScript Object Notation) library that is included in most browsers. It takes any Javascript object and converts it to one big string. If the object is an array, as here, all of the objects in the array are included in the resulting string. Since it is one string, we can simply save it to localStorage in one step.

Of course the difficult will come in restoring the complete list of Appointment ***objects*** when we reload the data. Since each appointment has been converted to an equivalent string, it will have lost the internal data representations (e.g. of the datetime field) and all of the methods associated with Appointment objects. When we load the array of appointments back from localStorage, we'll have to restore them as true Appointment objects.

5. Add the following function to the Javascript file:

```
var loadList = function(){
    var appts = "";
    if(localStorage.appointments !== undefined){
        appts = localStorage.appointments;
        appointments = JSON.parse(appts);
        var proto = new Appointment();
        for(var i=0; i<appointments.length; i++){
            var appt = appointments[i]
            // Attach the 'class' prototype to this object
            appt.__proto__ = proto;   //note __proto__ has double underscores either side
```

```
                // And make sure the datetime is actually a datetime
                appt.datetime = new Date(appt.datetime);
            }
        }
    }
```

This function first creates an appts string variable, then checks whether localStorage has an entry called appointments (i.e. if that enter is not undefined). If the entry does exist, the function loads the stringified version of the list of appointments into the appts variable, which we now have to convert back to a list of Appointment objects.

The JSON.parse() function takes a string and "de-stringifies" it – i.e. converts it back from a string to an object. In this case, it can manage to create an array of objects, but each object will have only string members. To convert each of these back to Appointments, all we need to do is re-establish the proper Appointment.prototype, and then convert the datetime elements back to proper Date objects. For this, we'll need a member of the Appointment type (proto) so we can get its prototype.

From here, step through each element in the appointments array, attaching the proto object to its __proto__ member (this is a Javascript 'hidden' member that links to the Appointment.prototype definition. Finally, use the current datetime member of each appointment (now a string) to create an actual Date object, which is assigned back to overwrite the string version. We have now reconstructed the entre list of appointments.

6. Finally, arrange for loadList to be called when the page loads (i.e. place a call to it in window.onload, just before the call to showTable(). You can test the app by creating a couple of appointments as usual, and then closing the browser. Re-load the app into the browser and you should find that the table of appointments is automatically restored.

Exercises

1. There are (at least) two annoyances in this app: the list of appointments just gets bigger, even when the appointments are completed, and once an appointment is due, it will keep alerting you every time the timer fires. Fix these problems:
   a. The first problem could be fixed by filtering out overdue appointments in the loadList() function. Easiest is to only add an appointment to the list if it is NOT due
   b. The second can be dealt with be marking an appointment as completed (i.e. setting .completed to true) once it becomes due, and then only alerting appointments if they are not marked as completed.
2. As a bit of advanced functionality, we could arrange for the user to re-schedule an isDue alarm whenever one is fired. Instead of using a simple alert() function, you can instead call the confirm() function (which gives options for Yes/No, True/False input with two buttons). Pop up an confirm message something like:

```
if( appointments[i].isDue()){
    if(confirm(appointments[i].subject + "\n" +
                    "Do you want another reminder in 15 minutes?"){
        // here, we would adjust the appointment time by adding 15 min
    } else
        // Just cancel it…
        appointments[i].completed = true;
    }
}
```

End of Lab 4