COMP3006 – Full-Stack Development

Workshop 4 – Local Storage and Concurrency

Autumn 2020

This week's lecture was on local storage and concurrency in JavaScript, and this workshop will explore those topics further.

You will also extend your work on the mini project by adding local storage to the chat application you've been developing.

Before the Workshop

Exercise 1

Write a simple web page that checks local storage for the module name and module number on page load, and displays it in the page if it is found. You should provide a pair of text inputs to enter this information, and a button that when clicked saves the data to local storage.

When you have written the program, test it by loading it in your browser. On the first go you shouldn't see the data presented in the page as you haven't added it to local storage yet. On the second go you should see the values you have entered, like so:

Exercise 2

Implement a hello world application as was done in the lecture.

- Download the Node webserver that is available on the DLE. This must be in the same directory as the code you write for this workshop (it's only needed for the web worker parts, not for the local storage).
- Create a HTML file called **helloww.html** that has a paragraph element with the ID **#messagePar** (give it the contents "Result:" or similar).
- Add a script element to the HTML page and modify it as follows:

```
$(function() {
  // Define the worker and point at the helloWorker.js file.
  let worker = new Worker("/helloWorker.js");

  // Process the response message sent by the worker.
  worker.addEventListener("message", function(evt) {
    // Extract the data from the event - this is the message,
    // put it into the page using jQuery.
    let msg = evt.data;
    $("#messagePar").append(msg);
  });

  // Invoke the worker.
  worker.postMessage("David");
});
```

- Create a new file called **helloWorker.js** and modify it as follows:

```
self.addEventListener("message", function(evt) {
  // Extract the data from the event, store in a variable called
  // name, and use it to construct a message.
  let name = evt.data;
  self.postMessage("Hello " + name + " from a web worker");
});
```

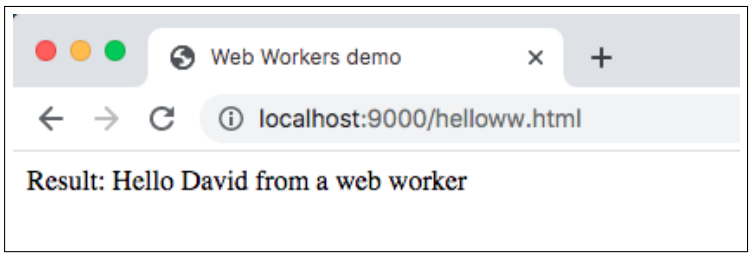
As discussed in the lecture you need to serve these pages from a webserver – as we haven't covered Node yet I've provided you with one that you can run. You need to make sure you name your files correctly – **bubble.html**, **main.js** and **sortWorker.js**. The Node server needs to be in the same directory as these files.

Change my name to yours!

- Open a terminal – such as **Command Prompt** in Windows, and use the **cd** command to change directory to the location of your code. Then run **node server.js** to activate the server.

If you're using your own machine for this you may need to install NodeJS to do this part.

- Browse to **http://localhost:9000/helloww.html** and you should see the following in your browser:



During the workshop

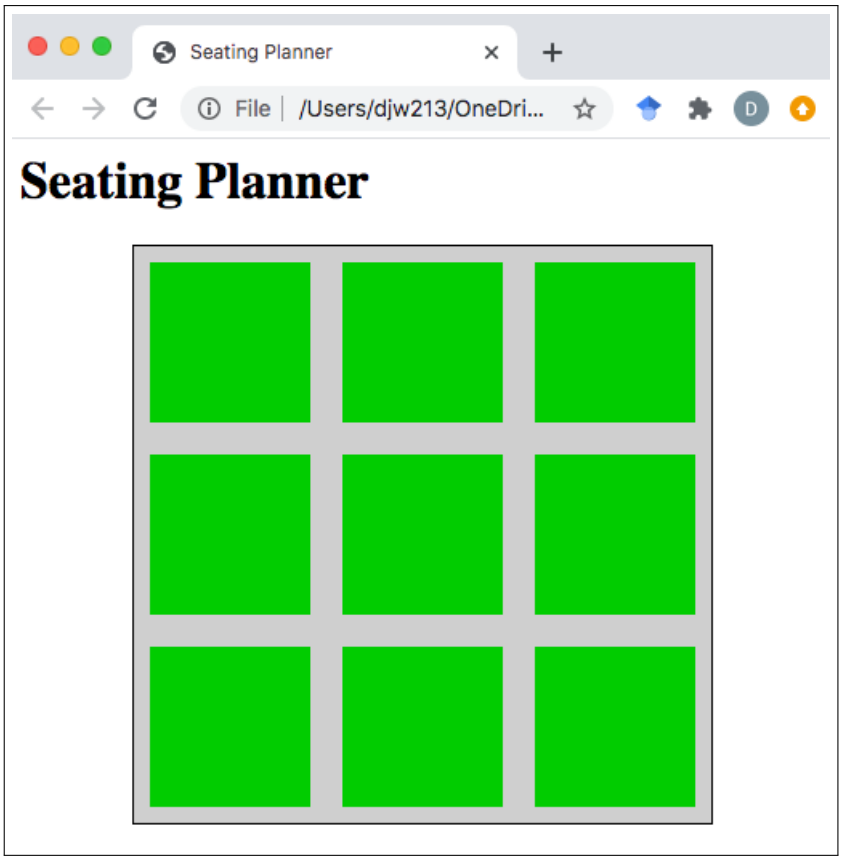
Exercise 3

Implement the client for a room booking application that uses local storage to recall which seats are booked.

- Create an HTML page that contains a level-one heading **Seating Planner**. Below that should be a div with ID **room**, within which are 9 divs of class **seat** with IDs **seat-01**, **seat-02**, etc.
- Add some CSS to style your elements:

  1. Add an declaration block for the **#room** element that:
     - Applies a background colour of #CFCFCF;
     - a 1px black border;
     - a width of 360px;
     - uses the **margin** and property to centre the element using the **auto** value;
     - and uses the **overflow: hidden** property to make sure the seats are contained within the **#room** div.
  2. Add a declaration block for the **seat** class that:
     - Applies a **width** and **min-height** of 100px;
     - a margin of 10px;
     - floats the elements to the left;
     - and applies a background colour of #00CC00.
  3. Add a declaration block for the **booked** class that sets the background colour to #CC0000.

  When you have finished the CSS you should have something that looks like this:



- Once you have completed the elements of the user interface and styled them you can start building the JavaScript portion of the program. Add style tags, and an onload event handler. Within the event handler:

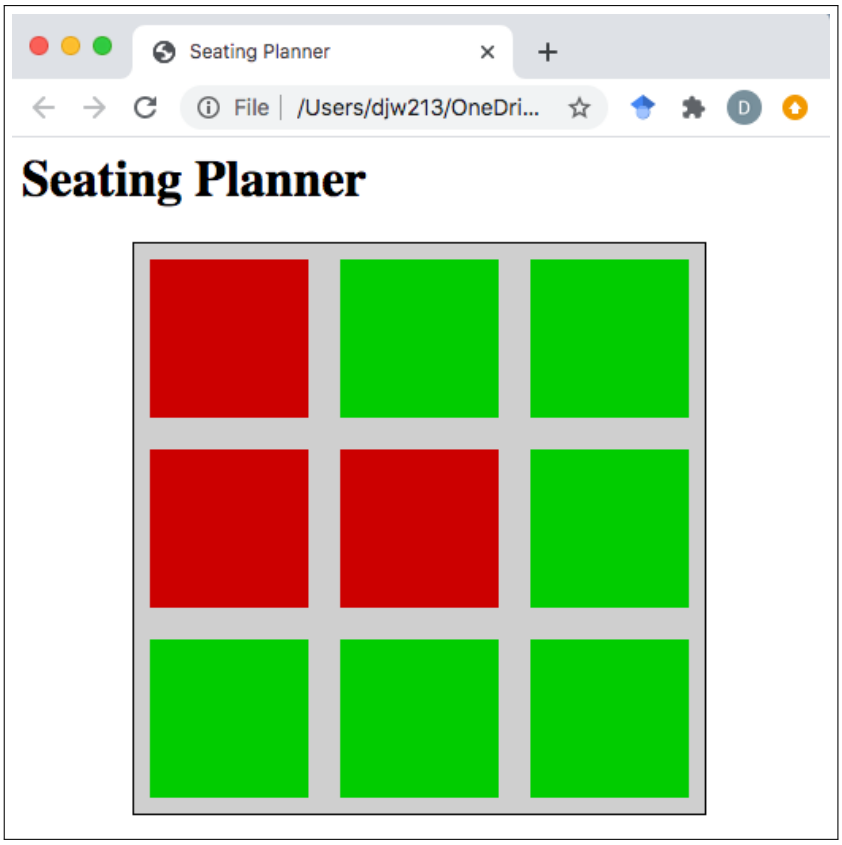
  1. Add an event handler to the **seat class**. This should
     - Either add or remove the **booked** class to the specific seat.
     - If the seat has the class **booked** then use the **localStorage.setItem** method to add the seat Id (**this.id**) to the local storage.
     - Otherwise, if the seat does not have the **booked** class, remove the seat from local storage with **localStorage.removeItem**.
  2. As the program loads, loop over the possible seat IDs and check to see if the **localStorage.getItem** method returns **null** for each key, and add the **booked** class if not.

  Having completed the JavaScript code you will have something that looks like this (PTO). Try setting some seats, closing the tab and reloading it. If the seats you clicked on are still shown in red then your program is working properly.

I suggest you jQuery to do these exercises.

Within the event handler you can use the selector **$(this)** to refer to the specific seat that has been clicked on.

The jQuery method is used **toggle** method to change the class and **hasClass** to check if the element has the class.



Exercise 4

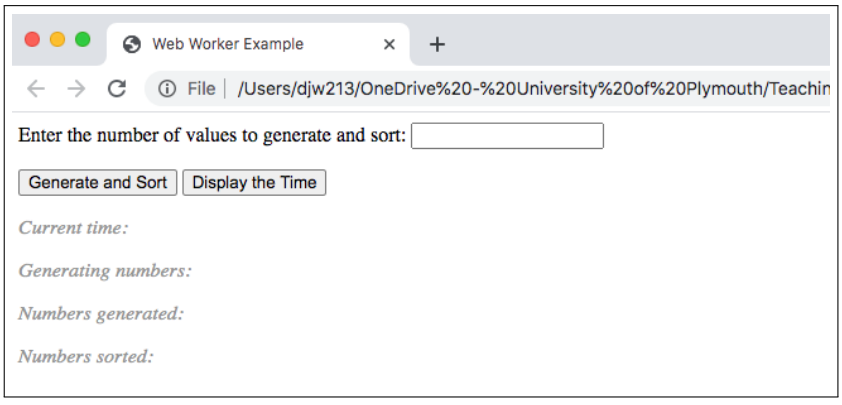
Implement a client-side application that sorts a given number of values with the bubble sort algorithm, using web workers.

- Download the Node webserver that is available on the DLE.
- Begin by creating a HTML page that has a text box and associated label to take a numerical value, a pair of buttons, and four paragraphs.

  The four paragraphs will be used to store status updates – apply some styling to them so that they look different to the input label.

  Include an empty paragraph at the bottom with the ID **results**.

  The page should look something like this:



- Link a JavaScript file (**main.js**) to the page. Add an event handler to the date button that generates the **current timestamp** and display that timestamp in the **current time** paragraph.

- Implement the **bubble sort** algorithm as a JavaScript function.
- Attach an event handler to the **generate and sort** button. The handler should:
  1. Extract the value from the input field and generate an array of values from 0 up to the number specified.
  2. Call the bubble sort function, passing the array of values.
  3. Display the sorted list in the results paragraph.

  As your program executes it should log the following times and display them in the page:

  – Prior to generating numbers;
  – After numbers have been generated;
  – After the numbers have been sorted;

  Once you have completed these steps you can test the program. **You should use a small number to start with** as the aim of this exercise is to block the browser. I suggest you test with the number 10.

  Once you have tested the program and it works you can try a larger number. Once you are generating and sorting the numbers, try clicking the **Display the Time** button, and you will see that the browser is blocked.

This was one of the examples in Lecture 2, available on the DLE as a video.

Be careful here! If you choose too large a number you'll loose your browser tab for a good while. 50,000 worked for me, but your computer may work at a different speed. I suggest you work up to that, and go beyond if necessary.

The next step is to refactor the code so that it uses a Web Worker. The aim of doing this is to retain the use of the UI while the numbers are sorted.

- Begin by starting a second JavaScript file in the same directory as your HTML and existing JS code. Call it **sortWorker.js**.
- Refactor the **main.js** file. Remove the call to the **bubble** function and follow the pattern from Exercise 2 to replace it with:
  1. Initialise a web worker object (point it at **sortWorker.js**).
  2. Add an event handler to process the **message** event that fires when the web worker sends back its response. The response will be an array, which should be added to the **result** paragraph, and the timestamp added to the final status ("Numbers sorted") paragraph.
  3. Use the **postMessage** method on the worker object to send the array of numbers to the worker.

Once you have completed the modification to the code so that it uses the worker you again sort your large array of numbers. This time, you should retain use of the UI while the sort executes.

Mini Project

This week you will extend the mini project by adding what you have learned this week about local storage in JavaScript.

Exercise 5

Add local storage to the chat application. Modify the page so that as the user types their message is added to local storage. In the event of the chat crashing, their message should be saved when they browse back to the page.

Extension task

Exercise 6

Alter the seating planner from Exercise 3 so that it uses **PouchDB** instead of local storage. You will need to install Pouch locally on your device, and should implement the application so that if the connection is lost the data is mapped to a remote instance of the database.

By “remote”, I mean that it can be running on your laptop, but should be accessed using a **localhost** URL rather than using it locally in the browser.