ELECTRICAL AND INFORMATION ENGINEERING
University of the Witwatersrand, Johannesburg
Software Development III

Laboratory 3 — Front-End Web Development

This lab provides an introduction to front-end web development using RESTful interfaces, Asynchronous HTTP Requests and Bootstrap.

1 Front-End Development

We are first going to focus on front-end development. To simplify the process of local development, we will install the Live Server package from the npm package manager. Live Server is a development server which has live reload capabilities. This means that your web page is reloaded as you work, which is extremely useful for viewing any changes you might have made, or for interacting dynamically with your web page.

Start by creating a new folder for your project. Next, install Live Server from the npm package manager as follows:

```
npm install -g live-server    # -g: install package globally
```

Inside the folder, create a file called `index.html` which will be the page that is served by your server. Insert the following into the file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Class List</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>Students in the School of EIE:</h1>
</body>
</html>
```

Now, it is time to run your server. In order to run live server, you need to run the `live-server` package in the folder that contains your HTML and JavaScript code. In your terminal, navigate to the folder where your HTML and JavaScript code is saved. Then type the following:

```
live-server
```

This should start your local server with `index.html` being served from http://127.0.0.1:8080 in your browser. Edit `index.html`, and notice how when you save, your web page is automatically updated.

To close the server, type Ctrl C into your terminal window.

1.1 Manipulating the Document Object Model

Now that we have a basic web page, lets learn how to dynamically add elements to the page.

Hint: If you need any help with front-end development, make use of the tutorials on W3Schools.

1.1.1 Adding basic elements

Edit `index.html`:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Class List</title>
    <meta charset="utf-8" />
    <!-- defer means that the JS code is executed after the DOM is fully built -->
    <script defer src="index.js"></script>
</head>
<body>
    <h1 id="heading">Students in the School of EIE:</h1>
    <button type="button" id="addButton">Add</button>
    <div id="studentList"></div>
</body>
</html>
```

Here, we are linking our html file to the JavaScript script that will be dynamically changing our web page.

Now, create `index.js`.

In `index.js` we want to create a function called `addParagraph()` which adds paragraph elements containing text to the `div` under the button.

```javascript
const button = document.getElementById('addButton')

button.addEventListener('click', function () {
  const paragraph = document.createElement('p') // Create <p> element
  const text = document.createTextNode('This is a student') // Create text node
  paragraph.appendChild(text) // Append the text to <p>
  document.body.appendChild(paragraph) // Append <p> to <body>
}, false)
```

Let's unpack the code above. All browsers create a global object called `document` which represents the *Document Object Model* (DOM) that contains the structure of the main `html` file. It contains all of the HTML elements as children as well as useful functions in JavaScript. By calling `getElementById()` the object returns the first element in the DOM with the provided ID. Note that HTML ID's should be unique.

Every element in the DOM can be assigned *event listeners* which are called when a specific event occurs. For our button we assign an anonymous function to the *click* event. You can add multiple listeners to the same element and the same event.

If you assign a listener to a parent and child element for the same event, the sequence they are called in can either be `capturing` or `bubbling`. Most of the time you can default to `bubbling` by setting the third parameter of `addEventListener()` to `false`. If your listeners need to be called in a specific order look at how modern browsers order events.

In this function, we create a new paragraph (`<p>`) element using the `createElement()` function. We then create a text node and append it to the paragraph using the `appendChild()` function. Now that we have created our paragraph, we append it to the body of our HTML document.

Whenever the button is pressed, the function `addParagraph()` is called, and the new paragraph is added to the body.

Exercise 1

Dynamically add the following elements to the page: a level two heading and a hyperlink.

1.1.2 Interacting with the elements on the web page

We can now add elements to the body of the document. However, normally, we will want finer control over where we add our elements. In HTML, each element can be assigned a unique ID. This ID can then be used to select that specific element using the `getElementById()`. For example, if we wanted to add a paragraph to the `<div>` in our HTML code, we would do this as follows:

```
const paragraph = document.createElement('p')
const text = document.createTextNode('This is a student')
paragraph.appendChild(text)
const students = document.getElementById('studentList')
students.appendChild(paragraph) // Append <p> to the <div>
```

We can also edit elements dynamically using their IDs. Lets change the page heading upon pressing the button.

Change `index.js` as follows:

```
const button = document.getElementById('addButton')

button.addEventListener('click', function () {
  const headerElement = document.getElementById('heading')
  headerElement.innerHTML = 'My New Heading'
}, false)
```

In this function, we find the `h1` element using its ID and then set the HTML content inside the element to our new string.

Exercise 2

Here, you will need to make use of the `students` array given in Listing 16 of Lab 2. Add a button to your webpage that displays a list of all of the students in the array. Each student in the list should also have a button that allows for the editing of that student's details. A button that deletes the entire list should be present at the bottom of the list. This button should also delete itself when the student list is deleted.

1.1.3 Creating a class list filter

Being able to edit elements on the page is fun, but we haven't produced anything useful yet. Lets work towards creating a class list, where we can filter students by their name, student number, or year of study. We want to implement live filtering whereby as you type, the students whose names do not match the search text are filtered out.

Here, you will make use of your solution for Exercise 10 of Lab 2.

Edit `index.html` to look like the following:

```html
<!DOCTYPE html>
<html>

<head>
  <title>Class List</title>
</head>

<body>
  <h1>Class List</h1>
  <!-- 'input' is an empty element, it consists of single tag -->
  <input type="text" id="search-text" placeholder="Filter students">
  <div id="students">
    <!-- Use this div to hold the list of students -->
  </div>

  <script src="class-list-app.js"></script>
</body>

</html>
```

In the HTML code, the `<input>` tag specifies that there must be an input field where the user can enter text. In this case, the input field is where the user will enter their search term.

In order to perform the live filtering, we need to call a JavaScript function every time the text in the input field changes. We will do this using the event listener in Listing 1.

```javascript
document.querySelector('#search-text').addEventListener('input', function(e){
    // DO SOMETHING
})
```

Listing 1: Event Listener that fires on every change

The `querySelector()` method returns the first element that matches a specified selector in the document. In this case, it returns the first element with the ID `search-text`. Earlier you will recall that we used the `getElementById()` function to query the DOM. In this case, you can use either method. `getElementById()` can only be used for retrieving an element by its ID, while `querySelector()` will match any selector including CSS classes. You can use `querySelectorAll()` to retrieve all elements having a particular class.

The `input` function indicates that the event fires on every change of the element with the selected ID. In other words, the function will execute after each letter is typed.

Exercise 3

Using the filter function you created in Lab 2, and your new knowledge of dynamically adding HTML elements using JavaScript, create a class list web page where you can perform a live search of the students by their name.

The JavaScript code to do this must reside in the file: `class-list-app.js`. The general approach is to filter the `students` array based on the user's search text and then iterate over the filtered array, appending each element to the `div`.

Exercise 4

Extend your previous solution and allow the user to select what property they want to filter by (Hint: Create a dropdown menu with the filter options).

Note: When implementing client-side functionality in JavaScript you should ensure that your code is compatible with the browsers your clients intend to use.

1.2 Cookies

Cookies are data that is stored on your computer in small text files. Cookies are primarily used to store and recall information about a user, especially by servers. By using the local storage of the computers, resource usage can be offloaded to the client and factors such as dynamic IPs do not create problems with lost information and conflicts. Cookies are sent with the request for a page from a server and are one of the technologies used to maintain user *sessions* which is why you can log on to websites once with a browser and not have to log on again for every page that you visit during your session on the site.

Cookies can be created and read using JavaScript. The cookies can be created using the DOM with the use of the `document.cookie` property. Cookies are stored in name-value pairs similar to data in objects, as follows `username= john447`.

Cookies are created in the following manner:

```
document.cookie="username=john447"
```

Cookies are deleted by default whenever the browser closes; however, they can also be set with a specific expiry date and time, in UTC. A path parameter can be specified to identify which page the cookie belongs to, as shown below. This example assigns the path to the current page:

```
document.cookie="username=john447;expires=Fri,28 Aug 2020 12:00:00 UTC;
path=/"
```

A cookie can be read using the same property that created it (`const x=document.cookie`). This returns all the cookies in one string. The same property can be used to change or delete cookies by setting the cookie to something else or setting the expiry date to a time that has passed, respectively. The cookie path must be set in order to change or delete it and changing it requires the name to be the same. Adding a cookie is done by setting `document.cookie` to the value of the new cookie. If the new cookie does not have the same name and path as an existing cookie, it is appended to the list of existing cookies.

Exercise 5

On your `index.html` page, create a field that allows a user to enter their name and store that in a cookie named `UserName`. Add a function that reads the cookie adds the text "Hello <UserName>" on page load, where UserName is the name extracted from the cookie. Set the cookie's expiration date to the end of the year and ensure the cookie, and thus the greeting, is preserved when the browser is closed and opened again.

2 Back-End Development

Now that we have an idea of how to perform front-end development, let's focus on the back-end. Let's make a web site to manage the class lists for the School of Electrical and Information Engineering.

We will be working with the source code from Section 14 of Lab 2.

2.1 Creating the Router

Create a new folder for this part of the lab, and install Express (as in Lab 2). Within this folder create a new routes file called `classRoutes.js` to serve our new website with the following in it:

```
const path = require('path')
const express = require('express')

const router = express.Router()
const classList = [] // our class list array
router.get('/', function (req, res) {
  res.sendFile(path.join(__dirname, 'views', 'class', 'index.html'))
})
router.get('/create', function (req, res) {
  res.sendFile(path.join(__dirname, 'views', 'class', 'create.html'))
})
router.get('/delete', function (req, res) {
  res.sendFile(path.join(__dirname, 'views', 'class', 'delete.html'))
})
```

```
router.get('/edit', function (req, res) {
  res.sendFile(path.join(__dirname, 'views', 'class', 'edit.html'))
})
module.exports = router
```

Create the HTML files needed for the requests within a `views/class` folder. Type some text into the `index.html` so that you will know if is being displayed.

Create an `index.js` file, in the root folder, to mount your new router, with the following content:

```
const express = require('express')
const app = express()
// loading our routers
const mainRouter = require('./mainRoutes.js')
const classRouter = require('./classRoutes.js')
// mounting our routers
app.use('/', mainRouter)
app.use('/class', classRouter)
app.listen(3000)
console.log('Express server running on port 3000')
```

Lastly, copy the `mainRoutes.js` file from Lab 2 into the root folder as well.

Run the app and browse to `localhost:3000/class` and verify that your new router has been mounted to `/class` and returns your `index.html`.

2.2 RESTful Interface

In order to actually manage our class list, we will be implementing a RESTful interface. Add the following to `classRoutes.js` (in the appropriate place):

```
// RESTful api
router.get('/api/list', function (req, res) {
  res.json(classList) // Respond with JSON
})

router.get('/api/get/:id', function (req, res) {
  res.json(classList[req.params.id]) // Notice the wildcard in the URL?
  // Try browsing to /api/get/0 once you've added some entries
})

router.post('/api/create', function (req, res) {
  console.log('this will create a student entry')
  res.redirect(req.baseUrl + '/create')
})

router.post('/api/delete', function (req, res) {
  console.log('this will delete a student entry')
  res.redirect(req.baseUrl + '/delete')
})
```

```
router.post('/api/edit', function (req, res) {
  console.log('this will edit a student entry')
  res.redirect(req.baseUrl + '/edit')
})
```

Now we have routes for POST requests, the most common types of these are form submissions. In order to read the data sent in a POST request, we need to use the `body-parser` npm module. Install `body-parser` using npm and edit `index.js` to look like the following:

```
const express = require('express')
const app = express()

// loading body-parser
const bodyParser = require('body-parser')
// loading our routers
const mainRouter = require('./mainRoutes.js')
const classRouter = require('./classRoutes.js')

// tell Express to use bodyParser for JSON and URL encoded form bodies
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({ extended: true }))

// mounting our routers
app.use('/', mainRouter)
app.use('/class', classRouter)

app.listen(3000)
console.log('Express server running on port 3000')
```

Launch the server and browse to `localhost:3000/class/api/list` and confirm that it is returning an empty array.

2.3 Creating Forms for POSTs

Now that we have a RESTful interface, let's create some forms so that we can add, delete and edit the students in the class.

Edit `create.html` to be:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Class List: Create</title>
    <meta charset="utf-8" />
</head>
<body>
    <form action="api/create" method="post">
    <div>
        <label for="studentName">Enter The Student's Name:</label>
        <input type="text" id="student" name="student">
    </div>
    <div class="button">
        <button type="submit">Add</button>
    </div>
</form>
</body>
</html>
```

Launch your server and try adding a student. Check Node's console output.

Now, let's edit our RESTful endpoint to actually add a new student to the classList. Update the routing code for /api/create as follows:

```
router.post('/api/create', function (req, res) {
  console.log('Adding the following student:', req.body.student)
  classList.push(req.body.student)
  res.redirect(req.baseUrl + '/create')
})
```

Try adding a student using your form again, then view the student array.

Exercise 6

Add forms and routing code for edit and delete. You will need to send the ID of the student that you want to edit or delete.

3 Client-Side Rendering and Single Page Applications

Now we have a working class list app, but browsing to different pages every time we want to add, remove or edit an entry is really inconvenient. Let's make a page that can do all these things without reloading. These types of web pages are called "Single Page Applications" or SPA's and they use JavaScript in the browser to do processing and communication with the server.

3.1 Serving Static Files

In most websites, we will need to serve some files directly to the clients. Usually, these are things like images, stylesheets and scripts that need to be loaded on the client side.

While we can add these as routes the same way that we created our webpages, this can quickly become tedious. Instead, Express provides a piece of middleware called static that allows us to mount a directory at a path and serve all files inside it as if they were routed. Let's do this so that we can serve our scripts.

Firstly, create a folder in the root of your project called public and inside it create folders for scripts, images, libraries and css. Now add the following line to your server's index.js, after app.use('/class', classRouter), to serve the files:

```
app.use('/cdn', express.static('public')); /* this will mount
your public directory to '/cdn'. i.e. your scripts folder
will be at /cdn/scripts */
```

3.2 Asynchronous HTTP Requests using Fetch

Now that we can run code in response to events, we can interact with our RESTful endpoints using `Fetch`. Essentially, this involves using client-side JavaScript to make requests to the server and processing the responses in the background on the client.

`Fetch` is a modern alternative to `XMLHttpRequest` which is used for AJAX. `Fetch` is fully supported by modern browsers. The function `fetch()` returns a *promise*. Promises greatly simplify the management of asynchronous tasks as the flow of function calls is made explicit and readable. Promises will be covered in a future lab/lecture. As you can see from the lack of any parameters beyond the URL of the endpoint, the default behaviour of `Fetch` essentially looks for a GET endpoint. If you want to use `Fetch` for other endpoints, like POST, you have to specify this in the second parameter of the `fetch` function, as you can see below.

```
fetch(url, {
    method: 'post',//specify method to use
    headers: {//headers to specify the type of data needed
        'Content-Type': 'application/json'
    },
    body: JSON.stringify(data) // fill body of request. Here the data is a JSON object
})
.then(function(response) {}
    if(response.ok)
      return response.json(); // Return the response parse as JSON if code is valid
    else
        throw 'Failed!'
}).catch(function (e) { // Process error for request
    alert(e) // Displays a browser alert with the error message.
     // This will be the string thrown in line 7 IF the
    // response code is the reason for jumping to this
    // catch() function.
})
```

Now let's test this out. First create `class/index.html` as follows:

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Class List</title>
    <meta charset="utf-8" />
    <script src="/cdn/scripts/class/index.js"></script>
</head>

<body>
    <h2>Students in the School of EIE:</h2>
    <ol id="classList">
        <!-- The list of students in the school-->
    </ol>

    <h3>Add a new student</h3>
    <form action="class/api/create" method="post">
        <div>
            <label for="studentName">Enter the student's name:</label>
            <input type="text" id="student" name="student">
        </div>
        <div>
            <button class="button" type="submit">Add</button>
        </div>
    </form>
</body>

</html>
```

Note that the accompanying script for this page is located at:
`public/scripts/class/index.js`

Create `public/scripts/class/index.js` with the following content:

```
// Remember, this code runs in the browser and retrieves data from the server

fetch('/class/api/list') // Returns a Promise for the GET request
  .then(function (response) {
    // Check if the request returned a valid code
    if (response.ok) { return response.json() } // Return the response parse as JSON
    else { throw 'Failed to load classlist: response code invalid!' }
  })
  .then(function (data) { // Display the JSON data appropriately
    // Retrieve the classList outer element
    const classList = document.getElementById('classList')

    // Iterate through all students
    data.forEach(function (student) {
      // Create a new list entry
      const li = document.createElement('LI')
      const liText = document.createTextNode(student)
      // Append the class to the list element
      li.className += 'student'

      // Append list text to list item and list item to list
      li.appendChild(liText)
      classList.appendChild(li)
    })
  })
  .catch(function (e) { // Process error for request
    alert(e) // Displays a browser alert with the error message.
    // This will be the string thrown in line 7 IF the
    // response code is the reason for jumping to this
    // catch() function.
  })
```

The last thing to do is to modify our router so that we return back to the index page after adding a student. In `classRoutes.js` change the redirect for `api/create` from

```
res.redirect(req.baseUrl + '/create')
```

to

```
res.redirect(req.baseUrl)
```

Start your server and try loading the page `http://localhost:3000/class`, if everything is working correctly, you should see an (initially empty) list of students which has been retrieved from the server. Add students and they should automatically appear on the list.

Exercise 7

Enhance your SPA so that you can also edit and delete the students using asynchronous HTTP requests to the server.

3.3 Bootstrap

Now that we have a working website, the only thing left to do is to make it look good. We handle the styling of our website using separate files called stylesheets. These allow us to define styles for certain elements and reuse them throughout our site.

Let's create a stylesheet in our public `css` directory called `main.css`. Put the following in it:

```css
body {
  background-color: powderblue;
}
.button {
  background-color: #4CAF50;
  border: none;
  color: white;
  padding: 15px 32px;
  text-align: center;
  text-decoration: none;
  display: inline-block;
  font-size: 16px;
  margin: 4px 2px;
  cursor: pointer;
}
#student {
  color: red;
  border: 5px solid black;
}
```

Import the stylesheet into your `class/index.html` with the following:

```html
<link rel="stylesheet" href="/cdn/css/main.css">
```

Open your the page in your browser to see how it works. Well, it's something.

Since designing stylesheets is difficult, we're going to use a pre-made stylesheet called Bootstrap. It was designed by Twitter to easily create good looking webpages.

First, download the CSS version of Bootstrap. Then replace the link to `css/main.css` with a link to Bootstrap.

Now let's try styling some of our elements. In `class/index.html`, apply the following classes to the Add button: `"btn btn-primary"`. Load the site and check how the button looks.

Exercise 8

Style the rest of your site using Bootstrap. Bootstrap features many different classes and features. Use different button and div classes to see the effects and which you prefer. More information can be found at the W3 Schools Bootstrap page.

Version 2.36, 2024 (`elen4010-lab3.tex`)