INFT 2202

Interactive – 6.2

Contents

Interactive - Week 6.2

Objective:

Practice with jQuery libraries

Learning Outcomes:

After conducting this exercise, the students should/will be able to:

a. Experience working with Ajax, native JavaScript Ajax and jQuery Ajax helper methods.

Workspace Instructions

1. Create a copy the previous lecture workspace folder, creating a new folder with an appropriate name
    a. Except **DO NOT** port/copy over the **node_modules** folder
        i. Within your new project folder (terminal) invoke the command: **yarn install**

2. Update the **package.json** to reference your folder/project name

3. Run your **lite-server** (project terminal) to ensure you are satisfied with the health of you project

header.html using Ajax

The objective if this section is to utilize Ajax to load the header context (navbar etc..) into every one of our pages. This would immediately reduce the code redundancy in each of our pages

1. Start by creating a new file called **header.html**
2. Copy the navbar (<**nav**>) from the **login.html** page, and paste it over to the **header.html**
    a. At this moment the only contents in your header.html should be just the nav bar

Injecting header.html into index.html page using Ajax

Now that we have a **header.html** with a **<nav>** component, let test injecting it, in our **index.html**.

1. First start by removing the **<nav>** element complete from the **index.html** page
    a. At this moment your index page when rendered should have no nav bar

2. We want to inject the navbar using JavaScript and Ajax, to start open up the **app.js** and refactor the **DisplayHomePage()**. These are the steps
    a. Step 1. Instantiate an XHR Object
    b. Step 2. Add event listener for the **readstatechange**
        i. The event handler should check for a **complete readystatus** and HTTP status code 200
        ii. If all is well (above), simply output the **responseText** to the console.
    c. Step 3. Open a connection to the server
        i. Input parameter: “GET”, “header.html”
    d. Step 4: Send the Request to the server
        i. Input parameter: null

```
//STEP 1: Instantiate an XHR Object
let XHR = new XMLHttpRequest();

//STEP2: Add an event listener for readystatechange
XHR.addEventListener("readystatechange", () => {

    if(XHR.readyState === 4 && XHR.status === 200)
    {
        console.log(XHR.responseText);
    }
});

//STEP 3: Open a connection to the server
XHR.open("GET", "header.html");

//STEP 4: Send the request to the server
XHR.send();
```

Review your console output to ensure your navbar is output successfully to the console.

```
App Started!
Home Page
<nav class="navbar navbar-expand-lg bg-body-tertiary navbar-dark bg-dark">
    <div class="container-fluid">
        <a class="navbar-brand" href="index.html"><i class="fa-solid fa-spider"></i> WEB6201</a>
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent" a
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarSupportedContent">
            <ul class="navbar-nav ms-auto mb-2 mb-lg-0">
                <li class="nav-item">
                    <a class="nav-link" href="index.html"><i class="fa-solid fa-home"></i> Home</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="products.html"><i class="fa-solid fa-project-diagram"></i> Products</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="services.html"><i class="fa-solid fa-map-signs"></i> Services</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="about.html"><i class="fa-solid fa-info"></i> About Us</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="contact.html"><i class="fa-solid fa-envelope"></i> Contact Us</a>
                </li>
            </ul>
        </div>
    </div>
</nav>
```

3. The next step is to inject the header into the index.html page
    a. Remove the **console.log()** we used initially to validate header output in the index page
    b. Utilize jQuery to retrieve the <header> html element from the index page, and inject the html navbar within it. The navbar should now be visible again.

4. The next step is to update the active link in the navbar (ie. **nav-link aria-current)** to correctly indicate the current page. We can achieve this by authoring jQuery selector, to select <a> elements that are descendants of <li> … if we additionally add a filter equal for “Home”, we can isolate the correct tag. Lastly add the “active” class, this is the way we can an active link on the navbar using jQuery.

```
XHR.addEventListener("readystatechange", () => {

    if(XHR.readyState === 4 && XHR.status === 200)
    {
        $("header").html(XHR.responseText);
        $("li>a:contains('Home')").addClass("active");
    }
});
```

5. Of course, the issue with our code right now, is we want to be able to reuse this for each and every link, so hard-coding “Home” is not ideal. Let’s refactor this a bit further to utilize string literal to use the value of the current pages <title> value

```
if(XHR.readyState === 4 && XHR.status === 200)
{
    $("header").html(XHR.responseText);
    $(`li>a:contains(${document.title})`).addClass("active");
}
```

6. Update all your page title to match the respective names of their link.

Injecting header.html into index.html page using Ajax

1. To make our Ajax calls more reusable, we need to export this functionality outside of our **DisplayHomePage()**. Let's create a separate function called **AjaxRequest** that:
    a. Accepts three parameters: **method**, **url** and **callback**
    b. Copy the contents of the original Ajax call (Step 1 - 4) and relocate them to be inside the **AjaxRequest()** function instead of the **DisplayHomePage()**.
    c. Refactor the code in **AjaxRequest()** to utilize is passed parameters instead, and thus become more generic.
    d. Add a line within the code, that validates the callback passed is actually a function (refer to textbook). If the callback passed is not a function, issue a console.log error.
    e. Refactor the **DisplayHomePage()** to call the **AjaxRequest()** function now.

```
function AjaxRequest(method, url, callback){

    //STEP 1: Instantiate an XHR Object
    let XHR = new XMLHttpRequest();

    //STEP2: Add an event listener for readystatechange
    XHR.addEventListener("readystatechange", () => {

        if(XHR.readyState === 4 && XHR.status === 200)
        {
            if(typeof callback === "function") {
                callback(XHR.responseText);
            }else{
                console.error("ERROR: callback not a fucntion");
            }
        }
    });

    //STEP 3: Open a connection to the server
    XHR.open(method, url);

    //STEP 4: Send the request to the server
    XHR.send();
}
```

LoadHeader() Function

1. The **DisplayHomePage**() function call to **AjaxRequest**(), needs to be more generic, otherwise we are going to export the same amount of lines of code across all other Display functions that need to also make like-minded AjaxRequest() calls to obtain the same navbar. To remedy this, lets create a new function called **LoadHeader**().

```
function LoadHeader(html_data)
{
    $("header").html(html_data);
    $(`li>a:contains(${document.title})`).addClass("active");
}
```

2. Let's make the call to **LoadHeader**() down in our **switch** statement, so we only have to make the call once across all pages.

```
function Start()
{
  console.log("App Started!");

  AjaxRequest("GET", "header.html", LoadHeader);
```

3. Now, we can remove the legacy <nav>'s from each page as they are no longer needed.
    a. about.html
    b. contact.html
    c. contact-list.html
    d. edit.html
    e. index.html
    f. products.html
    g. register.html
    h. services.html

User Class Creation

In this next section of the class, we will explore the implementation of logging into our application. We start with the implementation of a user class.

1. Within the scripts folder create a user class, called **User.js** that meets the following requirements:
    a. The User class should be defined within a core name-spaced IIFE (similar to our **Contact.js**)
    b. The User class definition should have a constructor that accepts the following parameters:
        i. **Display Name**
        ii. **Email Address**
        iii. **Username**
        iv. **Password**
    c. Getters and Setters for DisplayName, EmailAddress, and Username,
    d. Overridden **toString()** method – recall what the intent of these are (**DO NOT** output password)

```
(function(core){

    class User
    {
        //constructor
        constructor(displayName = "", emailAddress = "", username="", password = "") {
            this.DisplayName = displayName;
            this.EmailAddress = emailAddress;
            this.Username = username;
            this.Password = password;
        }

        //overridden methods
        toString(){
            return `Display Name; ${this.DisplayName}\nEmail Address: ${this.EmailAddress}\nUsername: ${this.Username}`;
        }

    }
    core.User = User;

})(core || core={})
```

    d. Two utility methods for converting a User **to/from** - JSON format.
        i. toJSON()
        ii. fromJSON()

```
toJSON(){
    return {
        "DisplayName" : this.DisplayName,
        "EmailAddress" : this.EmailAddress,
        "Username": this.Username,
        "Password": this.Password
    }
}

fromJSON(data){
    this.DisplayName = data.DisplayName;
    this.EmailAddress = data.EmailAddress;
    this.Username = data.Username;
    this.Password = data.Password;
}
```

    e. A **serialize()** and **deserialize()** functions (we can copy these from our **Contact.js**). **Refactor** for our **User.js** class instead.

Mock User Data

Now that we have our User.js class in place we need to create some mock (test) data.

1. Create a **data** folder, that contains a **user.json** file with test user data. Ensure 1 entry is an **Admin** user.

```
{

  "users":
  [
    {
      "DisplayName": "Jane Doe",
      "EmailAddress": "janedoe@example.com",
      "Username": "janedoe",
      "Password": "123456"
    },

    {
      "DisplayName": "Admin",
      "EmailAddress": "admin@example.com",
      "Username": "admin",
      "Password": "789321"
    }
  ]

}
```

Login Page (login.html) Updates

We need to perform some updates to the login.html as outlined below.

1. Edit the login.html to add a message area to display error-oriented messages (login failure etc…)

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

2. Update the **contactName** text field to be **username** instead

```
<input type="text" class="form-control" id="username" name="username"  required
       value="" placeholder="Enter your username">
```

3. Update login.html to include **user.js** instead of contact.js

```
<script src="./scripts/namespace.js"></script>
<script src="./scripts/user.js"></script>
<script src="./scripts/app.js"></script>
```

4. Update the **button** on the login page:
   i. **id = "loginButton"**
   ii. **type = "button"**
   iii. Add fontawesome icon **class="fas fa-sign-in-alt"**

5. Create a **Cancel** button below the login button:
   i. **id = "cancelButton"**
   ii. **type = "button"**
   iii. **class = "btn btn-warning btn-lg"**
   iv. Add fontawesome icon **class="fas fa-undo"**

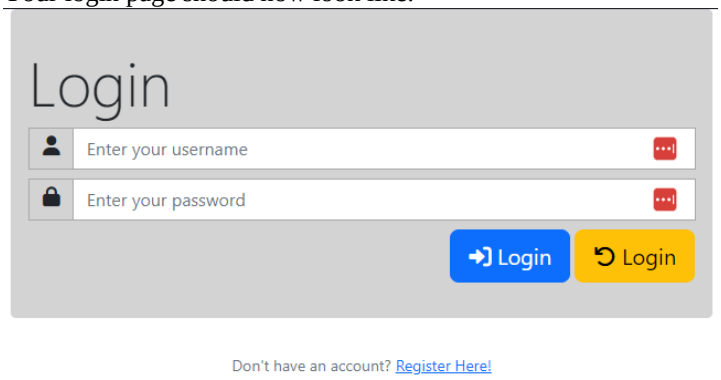
6. Within the **context/app.css** delete line 27 (width: 100%)

7. Add margins to the form-group (x 2)

```
<div class="form-group mb-2">
```

8. Change **text-right** to **text-end** (in ** ** near button)

Your login page should now look like:



Updates to DisplayLoginPage()

1. In your **DisplayLoginPage()** (within **app.js**) obtain a reference to the **messageArea** using jQuery (similar to our earlier lessons).

2. Initially hide the **messageArea** (**messageArea.hide()**) – as it should not be visible by default

3. Author an event handler for when the **#loginButton** is clicked, that:
    a. sets an initial **success** flag to **false**
    b. Instantiate a new user (**new core.User()**)
    c. Makes an Ajax call (get) to the **/data/users.json** data
        i. The **callback** handles checking, if the **username** and **password** match
        ii. If credentials match, sets the **success** flag (in step a.) to **true**, then breaks

```
$.get("./data/users.json", function (data){

    for(const user of data.users){

        //check if the username and password
        if(username.value === user.Username && password.value === user.Password)
        {
            newUser.fromJSON(user);
            success = true;
            break;
        }
    }
});
```

4. IF **success==true**,
    a. Add the new user (**serialized**()) to **sessionStorage**
    b. Remove the **class** attribute from the **messageArea**
    c. **Hide** the **messageArea**
    d. Redirect the user to the **contact-list.html**

    ELSE,

    a. Select the **username** textbox
        a. Change **focus** back to the **username**
        b. **Select** the data within the **username** textbox
    b. Add the following classes "**alert alert-danger**" to the **messageArea**
    c. Add an informative text to the **messageArea** indicating that login failed.

```
if(success){
    //add user to session storage
    sessionStorage.setItem("user", newUser.serialize());
    messageArea.removeAttr("class").hide();

    //redirect user to secure area of the site.
    location.href = "contact-list.html";
}else{
    // they do not match
    $("username").trigger("focus").trigger.("select");
    messageArea.addClass("alert alert-danger").text("Error: Invalid Login Credentials").show();
}
```

5. Next let's implement a cancel button, with the following responsibilities:
    a. **Reset** the login.html form
    b. **Return** to **index.html**

Logout Implementation

1. Let's first start by adding a link to the **header.html** page for the **login.html**. We will assign an **id** to this **<li>** element, so we can select it within our code (alter the link from login → logout, as needed).

```
<li id="login" class="nav-item">
    <a class="nav-link" href="login.html"><i class="fas fa-sign-in-alt"></i> Login</a>
</li>
```

2. Implement a **CheckLogin()** function, that has the following responsibilities:
    a. Check **sessionStorage** to see if a user has been stored.
        i. If true, select the login nav link (**id="login"**), and replace with a **logout**. There is **NO logout.html**, so just set to the **href = #**, for now

```
$("#login").html(`<a id="logout" class="nav-link" href="#"><i class="fas fa-sign-out-alt"></i> Logout</a>`)
```

    b. Develop the code to handle the case where the user clicks the logout button. Implement an event handler that simply clears the **sessionStorage**. Then redirect to **login.html**.

```
$("#logout").on("click", function(){
    //perform logout
    sessionStorage.clear();

    //redirect to login.html page
    location.href = "index.html";
});
```

3. Add the call to **CheckLogin()** to the **LoadHeader()** function. We must do this, due to the asynchronous nature of Ajax (in our case our AjaxRequest() function) – as we are not permitted to assume that the nav bar will be available otherwise.

AuthGuard Script

We want to develop a function to protect our **contact-list.html** so it is only visible to logged in users.

1. Create a separate file called **authguard.js** located in your **scripts** folder
2. Implement an **IFFE** that:
    a. Check if **sessionStorage** has a user
        i. If **NOT** true, forward to login.html

```
( function()
{
    if(!sessionStorage.getItem("user"))
    {
        location.href = "login.html";
    }

})();
```

3. Include a reference to the **authguard.js**, within the **contact-list.html** page BUT this time in the **<head>**, to ensure this is loaded as early as possible in the rendering process.

```
<!-- CSS Section -->
<link rel="stylesheet" href="./node_modules/bootstrap/dist/css/bootstrap.min.css">
<link rel="stylesheet" href="./node_modules/@fortawesome/fontawesome-free/css/all.css">
<link rel="stylesheet" href="./content/app.css">
<script src="./scripts/authguard.js"></script>
```

Update Public Repository

This concludes your ICE.

1. **Commit** and **push** all your code to your ICE **private repository.**
2. Make sure to **port/copy** your code changes over from your private repository workspace **to your public repository**.
3. **Commit** and **push** to your **public repository** (do this exclusively for each ICE only)
4. **Verify your github pages deployment.**

As a reminder I require the following as part of you ICE submission:

- Link (URL) to your **public** github pages site pasted in submission WYSWIG
- Link (URL) to your github **private** repo pasted in submission WYSWIG
- Copy zip of your source code uploaded to DCConnect (remove **node_modules**)