Seneca College

Web Programming

WEB322 – Week 9 – Class 1

WEB322 H

AJAX Review / Practical AJAX Programming

Up until this point, everything in this course has involved logic that exists primarily on the server (server-side). However, there piece of technology that plays a vital role on the client side that needs to be discussed as well. This is **AJAX (Asynchronous Jav and XML)** - a collection of technologies used together to create a richer user experience by enabling data to be transferred be web client (browser) and web server without the need to refresh the page.

AJAX Review: XMLHttpRequest

The XMLHttpRequest Object (often abbreviated to XHR) is what allows us to make requests for data once a page has loaded. It number of properties and methods that do the heavy lifting of establishing a connection to a server, monitoring the state of t request/response and executing custom code when the state has changed (ie, the request has completed and we have receiv response).

The following is a list of the most common properties / methods used when preforming an AJAX request on the client side (as **httpRequest = new XMLHttpRequest();**):

open()	**Method:** The **open()** method is responsible for initializing the request according to specific paramet which typically include the method ("GET", "POST", "PUT", "DELETE", etc.), and url responsible for ser data ("http://someurl/api/data", etc.). However there are a number of other optional parameters incl **async**, **user**, and **password**. A typical call to the **open()** method: `httpRequest.open('GET', 'https://reqres.in/api/users?page=1');`
setRequestHeader	**Property:** The **setRequestHeader** method is used primarily when we wish to **send()** data along with AJAX request (see send() below) and need to specify the "Content-Type". For example, if we wish to s JSON data, we would set the "Content-Type" to "application/json" `httpRequest.setRequestHeader("Content-Type", "application/json");`
	Method: The **send()** method is responsible for actually sending the request off to the location speci the "open()" method. If we are not sending any data to the server (ie: during a "GET" request) we do provide any parameters to the function. However, if we wish to send data, we would place it as a sin

send()

provide any parameters to the function. However, if we wish to send data, we would place it as a sin
parameter to the method, for example:

```
// send the request without any data
httpRequest.send();

// or send it with data
httpRequest.setRequestHeader("Content-Type", "application/json");
httpRequest.send(JSON.stringify({user:"John Doe", job:"unknown"}));
```

onreadystatechange

Property: The **onreadystatechange** property is an Event Handler that gets invoked whenever the **readyState** property of the XMLHttpRequest changes; for example, once the request has been initial completed. This often takes the form of an anonymous function, ie:

```
httpRequest.onreadystatechange = function(){
    // check the current value of the readyState & status properties
    // if successful, process the data
}
```

readyState

Property: The **readyState** property represents the "state" the the request is currently in (ie, 0: Unser Opened, 2: Headers Received, 3: Loading, 4: Done). We are typically interested when the request is "c (complete), however it is important to note that the request will move into this state whether or not request was successful.

For example, inside our onreadystatechange callback function we can find out if the current state is and if it is, we can check it's status, receive data, etc.

```
httpRequest.onreadystatechange = function(){
    // check the current value of the readyState & status properties
    // if successful, process the data

    if (httpRequest.readyState === 4) {
        // The request has completed ie, it's "done"
    }
}
```

Property: The **status** property represents the HTTP Status Code that was sent with the response. Id we want this to be 200 (Ok) - which states that the request has succeeded. We typically verify this in onreadystatechange callback function after it has been confirmed that the readyState is "done", for example:

```
httpRequest.onreadystatechange = function(){
```

status

```
        // check the current value of the readyState & status properties
        // if successful, process the data

        if (httpRequest.readyState === 4) {
            // The request has completed ("done")
            if (httpRequest.status === 200) {
                // We successfully received the response
            }
        }
    }
```

responseText

Property: Lastly, the **responseText** property holds the data returned from the request. Typically this JSON formatted string (see below) and we can create an object from this data using JSON.parse. Whe requesting data, we generally only wish to read the responseText if the response was completed successfully. To ensure this, we read the property only after we know that the **request has complete** the **status is 'OK'**, for example:

```
    httpRequest.onreadystatechange = function(){
        // check the current value of the readyState & status properties
        // if successful, process the data

        if (httpRequest.readyState === 4) {
            // The request has completed ("done")
            if (httpRequest.status === 200) {
                // We successfully received the response

                // Create object jsData from the JSON
                // received in responseText
                let jsData = JSON.parse(httpRequest.responseText);
            }
        }
    }
```

JSON Review

As we have seen from above, the format of choice when sending data to/from a webserver using AJAX is **JSON (JavaScript Obj Notation)**. This is a plain-text format that easily converts to a JavaScript object in memory. Essentially, JSON is a way to define using "Object Literal" notation, **outside** your application. Using the native JavaScript built-in JSON Object, we can preform the conversion from plain-text (JSON) to JavaScript Object (and vice-versa) easily. For example:

Converting JSON to an Object

```
let myJSONStr = '{"users":[{"userId":1,"fName":"Joe","lName":"Smith"},{"userId":2,"fName":"Jef
```

```
// Convert to An Object:
let myObj = JSON.parse(myJSONStr);

// Access the 3rd user (Shantell McLeod)
console.log(myObj.users[2].fName); // Shantell
```

Converting an Object to JSON

```
let myObj = {users: [{userId:1,fName:"Joe",lName:"Smith"},
                     {userId:2,fName:"Jeffrey",lName:"Sherman"},
                     {userId:3,fName:"Shantell",lName:"McLeod"}]};

let myJSON = JSON.stringify(myObj);

console.log(myJSON); // Outputs: '{"users":[{"userId":1,"fName":"Joe","lName":"Smith"},{"userI
```

Responding with JSON

Now that we're refreshed with the concept of actually making an AJAX request from the client side, why don't we set up a new server to respond to the request with some JSON? Fortunately, the Express.js framework makes this fairly simple with it's res.j method. In this case, we can pass the function a JavaScript Object and the method will automatically "stringify" it and send the converted JSON as the response. It will even include the correct Content-Type header value (ie: "application/json"). For exampl

```
const express = require("express");
const app = express();

const HTTP_PORT = process.env.PORT || 8080;

// setup a 'route' to listen on the default url path (http://localhost)
app.get("/", (req, res) => {

    let message = { msg: "Welcome!"};

    // send the message as JSON
    res.json(message);
});

// setup http server to listen on HTTP_PORT
app.listen(HTTP_PORT, () => {
    console.log("Express http server listening on: " + HTTP_PORT);
});
```

Creating a REST API (Introduction)

Creating a REST API (Introduction)

You may have heard of the term **REST** or **RESTful** API when reading about Web Programming. In fact, we have made use of a r
API (see the publicly available: https://reqres.in) to practice our AJAX skills by making requests for data. At it's most basic level,
essentially think of a REST ("Representational State Transfer") API as a way to use the HTTP protocol (ie, "GET", "POST", "PUT",
"DELETE", etc...) with a standard message format (ie, JSON or XML) to preform CRUD operations on a data source.

What makes this so valuable is that by creating a RESTful API, we remove any assumptions about how a client will access the d
could make HTTP requests to the API from a website, mobile app, etc. and it would be the website or app's job to correctly rer
data once it's received. This simplifies development of front-end applications that use the data and even removes any specific
programming language requirements for the client! If it can handle HTTP requests / responses and JSON, it can use our data!

Before we think about getting any kind of persistent storage involved however, why don't we first see how we can configure al
routes in our server to allow for CRUD operations on a simple collection of users in the format

```
{userId: number, fName: string, lName: string}
```

Route	HTTP Method	Description
/api/users	GET	Get all the users
/api/users	POST	Create a user
/api/users/:userId	GET	Get a single user
/api/users/:userId	PUT	Update a user with new information
/api/users/:userId	DELETE	Delete a user

When these routes are applied to our Express server code, we get something that looks like this:

```
const express = require("express");
const app = express();

const HTTP_PORT = process.env.PORT || 8080;

app.get("/api/users", (req, res) => {
    res.json({message: "fetch all users"});
});

app.post("/api/users", (req, res) => {
    res.json({message: "add a user"});
});

app.get("/api/users/:userId", (req, res) => {
    res.json({message: "get user with Id: " + req.params.userId});
});

app.put("/api/users/:userId", (req, res) => {
    res.json({message: "update User with Id: " + req.params.userId});
});

app.delete("/api/users/:userId", (req, res) => {
     res.json({message: "delete User with Id: " + req.params.userId});
});

// setup http server to listen on HTTP_PORT
app.listen(HTTP_PORT, () => {
    console.log("Express http server listening on: " + HTTP_PORT);
});
```

Here, we have made use of the request object's params method to identify the specific user that needs to be fetched, updated deleted based on the URL alone. In a sense, what we're allowing here is for the URL + HTTP Method to act as a way of querying data source, as **/api/users/3**, **/api/users/4923** or even **/api/users/twelve** will all be accepted. They may not necessarily return data, but the routes will be found by our server and we can attempt to preform the requested operation.

Now that we have all of the routes for our API in place, why don't we create a "view" that will make AJAX requests to test our A functionality? To begin, create a **views** folder and add the file **index.html**. This will be a simple HTML page consisting of 5 butt (each corresponding to a piece of functionality in our API) and some simple JavaScript to make an AJAX request.

However, since we are serving this file from the same server that our API is on, we will need to add some additional code to ou file; specifically:

```
const path = require("path");
```

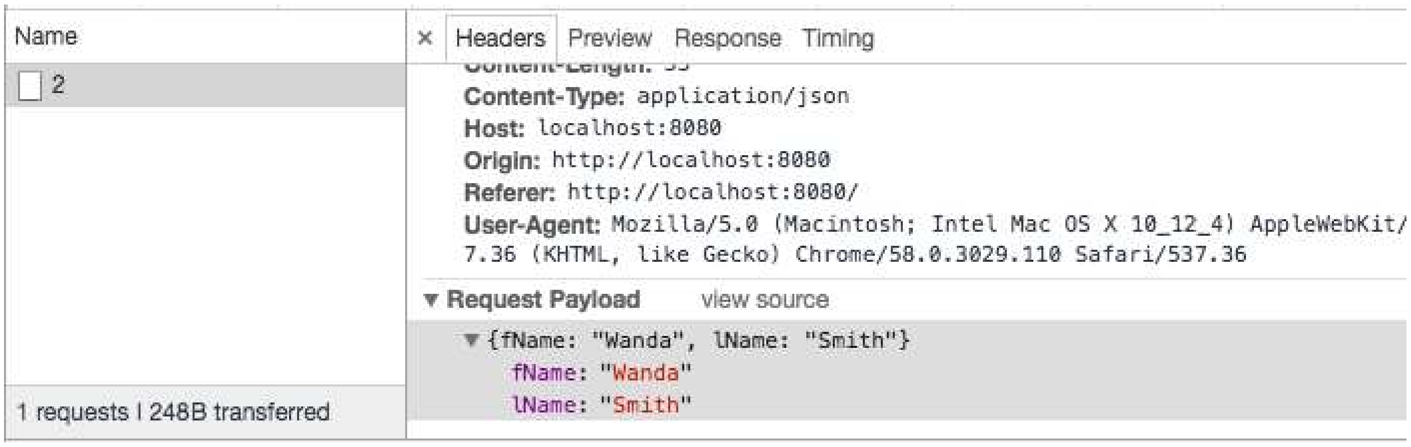
and

```
app.get("/", (req,res) => {
    res.sendFile(path.join(__dirname + "/views/index.html"));
});
```

Finally - our server is setup and ready to serve the index.html file at our main route ("/"). Our next step is to add our client-side JS to the index.html file. Here, we hard-code some requests to the API and output their results to the web console to make sur function correctly:

```
<!doctype html>
<html>
    <head>
        <title>API Test</title>
        <script>
            function makeAJAXRequest(method, url, data){
                let httpRequest = new XMLHttpRequest();
                httpRequest.open(method, url);
                if(data){
                    httpRequest.setRequestHeader("Content-Type", "application/json");
                    httpRequest.send(JSON.stringify(data));
                }else{
                    httpRequest.send();
                }

                httpRequest.onreadystatechange = () => {
                    if (httpRequest.readyState === 4) {
                        if (httpRequest.status === 200) {
                            console.log(JSON.parse(httpRequest.responseText));
                        }else{
                            console.log("Error getting Data");
                        }
                    }
                }
            }
            function getAllUsers(){
                 makeAJAXRequest("GET", "/api/users");
            }

            function addNewUser(){
                 makeAJAXRequest("POST", "/api/users", {fName: "Bob", lName: "Jones"});
            }

            function getUserById(){
                makeAJAXRequest("GET", "/api/users/2");
            }

            function updateUserById(){
                makeAJAXRequest("PUT", "/api/users/2", {fName: "Wanda", lName: "Smith"});
            }

            function deleteUserById(){
                makeAJAXRequest("DELETE", "/api/users/2");
            }

        </script>
    </head>
    <body>
        <p>Test the API by outputting to the browser console:</p>
        <!-- Get All Users -->
        <button type="button" onclick="getAllUsers()">Get All Users</button><br /><br />
        <!-- Add New User -->
        <button type="button" onclick="addNewUser()">Add New User</button><br /><br />
        <!-- Get User By Id -->
        <button type="button" onclick="getUserById()">Get User</button><br /><br />
        <!-- Update User By Id -->
        <button type="button" onclick="updateUserById()">Update User</button><br /><br />
        <!-- Delete User By Id -->
        <button type="button" onclick="deleteUserById()">Delete User</button>
    </body>
</html>
```

Once you have entered the above code, save the changes and try running your server locally to see what happens. You will se
of the routes tested return a JSON formatted message! Now we know that our REST API will correctly respond to AJAX request
by the client. Additionally, If you open the **Network tab** (Google Chrome) before initiating one of the calls to **Update** or **Add a**
User, you will see that our request is also carrying a payload of information, ie:



```
Content-Type: application/json
Host: localhost:8080
Origin: http://localhost:8080
Referer: http://localhost:8080/
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_4) AppleWebKit/
7.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36
▼ Request Payload
  ▼ {fName: "Wanda", lName: "Smith"}
      fName: "Wanda"
      lName: "Smith"
```

If we wish to capture this information in our routes (so that we can make the appropriate updates to our data source), we mu
some small modifications to our server.js file and individual routes (ie: POST to "/api/users" & PUT to "/api/users/:userId"). The
thing that we must do is include the body-parser module. Recall from week 5 - we require this module if we wish to access the
component of the request.

After this module is added (**npm install body-parser --save**), you can update the top of your server file to include the module:

```
const bodyParser = require('body-parser');
```

Additionally, we must include the following line before our routes, to ensure that our application can correctly access JSON for
data from the body of the request:

```
app.use(bodyParser.json());
```

Now that this change has been made, we can access data passed to our API using the req.body property. More specifically, we
update our POST & PUT routes to use req.body to fetch the new / updated **fName** and **lName** properties:

```
app.post("/api/users", (req, res) => {
     res.json({message: "add the user: " + req.body.fName + " " + req.body.lName});
});
```

and

```
app.put("/api/users/:userId", (req, res) => {
    res.json({message: "update User with Id: " + req.params.userId + " to " + req.body.fName +
});
```

If you have followed the instructions above correctly, your **server.js** file should look like this:

```
const express = require("express");
const path = require("path");
const bodyParser = require('body-parser');
const app = express();

const HTTP_PORT = process.env.PORT || 8080;

app.use(bodyParser.json());

app.get("/", (req,res) => {
    res.sendFile(path.join(__dirname + "/views/index.html"));
});

app.get("/api/users", (req, res) => {
    res.json({message: "fetch all users"});
});

app.post("/api/users", (req, res) => {
     res.json({message: "add the user: " + req.body.fName + " " + req.body.lName});
});

app.get("/api/users/:userId", (req, res) => {
    res.json({message: "get user with Id: " + req.params.userId});
});

app.put("/api/users/:userId", (req, res) => {
    res.json({message: "update User with Id: " + req.params.userId + " to " + req.body.fName +
});

app.delete("/api/users/:userId", (req, res) => {
     res.json({message: "delete User with Id: " + req.params.userId});
});

// setup http server to listen on HTTP_PORT
app.listen(HTTP_PORT, () => {
    console.log("Express http server listening on: " + HTTP_PORT);
});
```

If we try running our server and test our API again, you will see that the messages returned back from the server correctly ech
data sent to the API! We now have everything that we need to preform simple CRUD operations via AJAX on a data source usir
RESTful service. The only thing missing is the data store itself.

NOTE: If we want to allow our API to respond to requests from *outside* the domain (this is what https://reqres.in does), we will
enable Cross-Origin Resource Sharing (CORS) - see the 3rd party CORS middleware