LAYER ZERO

UNLV CYBER SECURITY CLUB

Exercise: Client Login with HTTP Auth

Goals

1. Listen for HTTP requests on an available, unused port
2. Create multiple backend routes
    a. resource that defines the URL
3. Allow multiple HTTP methods for the routes
4. Create a *login* route that allows POST methods in JSON format
5. Save a username and password via the *login* route
    a. make sure that:
        i. the server knows it’s receiving JSON and can correctly parse it
        ii. the client tells the server it’s sending JSON
6. Require this username and password in order to GET another route
    a. this time, include HTTP Basic Auth in the request instead of using a POST with data

Tools

- Node.js and node package manager (npm)
- text editor
- terminal / terminal emulator
- curl / Postman

Create & test a simple backend HTTP server

Setting up an *express node.js* server

- using npm, install the express framework

```
npm install -g express
```

- set up an express server

```
var express = require('express');
var app = express();
var port = 8080;

app.use(express.json());

app.listen(port);
```

- run the server

```
node myServer.js
```

Create some routes!

- also learn the smallest amount of javascript I’m so sorry
- briefly: *callback* functions
  - for the scope of this lesson, think of callback functions as just:
  - functions found within the parameter of other functions

```
var app = express();
...
app.get(‘/’);
// this defines the HTTP method and the corresponding route

function mainRoute(request, response) {
    response.send(“Hello world\n”);
}
// this is a function that takes in an HTTP request and HTTP response,
//  and returns some message in the response
```

Callbacks cont.

- but instead of *named* functions we can have *anonymous / lambda* functions

```
app.get(‘/’);

function mainRoute(request, response) {
    response.send(“Hello world\n”);
}
```

Callbacks cont.

- but instead of *named* functions we can have *anonymous / lambda* functions

```
app.get(‘/’);

function mainRoute(request, response) {
    response.send(“Hello world\n”);
}
```

```
app.get(‘/’,     );

function ~~mainRoute~~(request, response) {
    response.send(“Hello world\n”);
}
```

Callbacks cont.

- one more crazy thing you will see (ES6)

```
app.get(‘/’, function (request, response) {
    response.send(“Hello world\n”);
});  // notice that the unnamed funct is a param of the app.get() funct
```

Callbacks cont.

- one more crazy thing you will see (ES6)

```
app.get(‘/’, function (request, response) {
    response.send(“Hello world\n”);
});
```

```
app.get(‘/’, ~~function~~ (request, response) => {
    response.send(“Hello world\n”);
}); // we can remove the `function` keyword and add the arrow notation
```

Callbacks cont.

- one more crazy thing you will see (ES6)

```
app.get(‘/’, function (request, response) {
    response.send(“Hello world\n”);
});
```

```
app.get(‘/’, ~~function~~ (request, response) => {
    response.send(“Hello world\n”);
});
```

```
app.get(‘/’, (request, response) => {
    response.send(“Hello world\n”);
}); // final result
```

Okay for real now routes

- this will be the format of all our routes

```
app.get(‘/page’, (request, response) => {
    response.send(“Hello world\n”);
});
```

- but there is more than just HTTP GET right?
- create a few more app routes, send different response messages, use `console.log()` to view the request and response
- send requests with curl, etc.

```
curl -X OPTIONS http://localhost:8080/page
```

https://expressjs.com/en/4x/api.html#app.METHOD

Request, response parameters

- we can run `response.send(“string”)`
- but there is something else we typically associate with HTTP responses
    - maybe a detailed message about our request would be nice
    - but it would also be beneficial to receive some sort of code with this right

Request, response parameters

- we can run `response.send(“string”)`
- but there is something else we typically associate with HTTP responses
  - maybe a detailed message about our request would be nice
  - but it would also be beneficial to receive some sort of code with this right

```
app.get(‘/page’, (request, response) => {
    response.status(200).send(“Hello world\n”);
});
```

Request, response parameters

- we can run `response.send(“string”)`
- but there is something else we typically associate with HTTP responses
    - maybe a detailed message about our request would be nice
    - but it would also be beneficial to receive some sort of code with this right

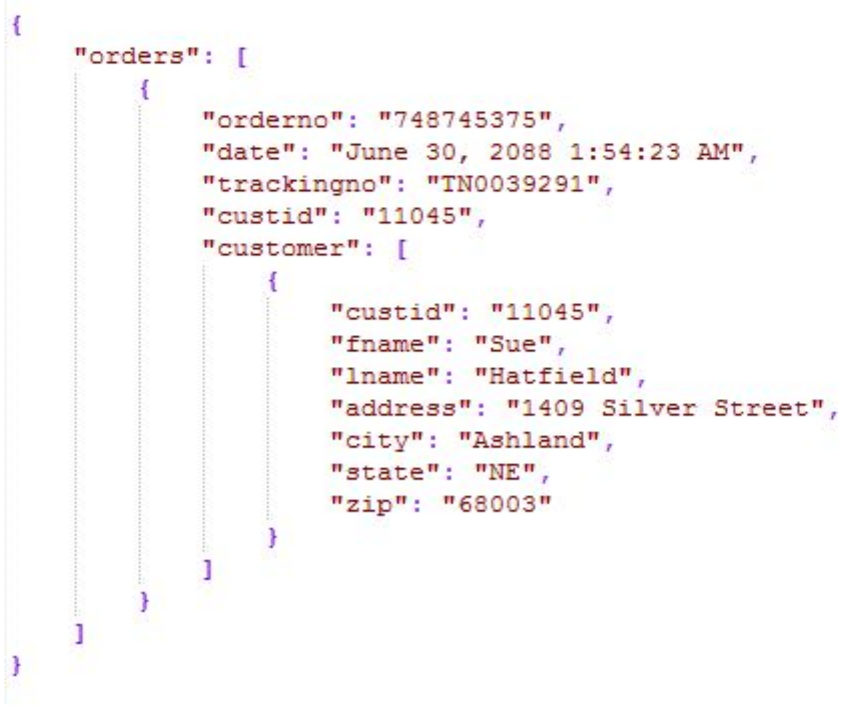
```
app.get(‘/page’, (request, response) => {
    response.status(200).send(“Hello world\n”);
});
```

```
curl -I -X GET http://localhost:8080/page
```

Send JSON data using curl & HTTP POST

JavaScript Object Notation (JSON)

- a way to format data under some standardization
- there are alternatives (e.g. YAML) and you can just make your own format if you want to



```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customer": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```

Quick check

1. ~~Listen for HTTP requests on an available, unused port~~
2. ~~Create multiple backend routes~~
3. ~~Allow multiple HTTP methods for the routes~~
4. **Create a *login* route that allows POST methods in JSON format**
5. Save a username and password via the *login* route
6. Require this username and password in order to GET another route

POST some data

- set up the server to send back the request that was sent to it

```
app.post(‘/login’, (request, response) => {
    response.send(request.body);  // notice we only care about the body
});
```

- attempt to send some JSON data

```
curl -X POST -d ‘{“username”: “myuser”, “password”: “hunter2”}’
http://localhost:8080/login
```

POST some data cont.

- remember that the client needs to tell the server what kind of data it’s going to send
- we specify the type of data using the headers

```
curl -X POST -H ‘Content-type: application/json’
-d ‘{“username”: “myuser”, “password”: “hunter2”}’
http://localhost:8080/login
```

HTTP Authorization

- we can specify a lot in the HTTP headers (content type of request, allowed type of response, etc)
- now we have to give some sort of authorization
  - the auth portion of the header includes some type of credential
  - these credentials can be a key/token, username and password, etc
  - for this example we will use **basic authorization**

```
Authorization: <type> <credentials>
```

Constructing a Basic Auth header

- the authorization portion of the header will look like this:

```
Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l
```

```
base64(username:password)
```

base64 encoding =/= encryption

- encoding makes sure no data is lost or modified during the transfer
- it is NOT for encryption, compression, security, whatever else

Utilizing authorization

- save username & password as a base64 encoded string when the user logs in

```
encodedAuth = Buffer.from(`${user}:${pw}`).toString('base64')
```

- send a request with a basic auth header to some “protected” route

```
curl --user username:password http://localhost:8080/admin/
```

```
curl -H “Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=”
http://localhost:8080/admin/
```

Goal Check

1. ~~Listen for HTTP requests on an available, unused port~~
2. ~~Create multiple backend routes~~
3. ~~Allow multiple HTTP methods for the routes~~
4. ~~Create a *login* route that allows POST methods in JSON format~~
5. ~~Save a username and password via the *login* route~~
6. ~~Require this username and password in order to GET another route~~

Outside User Exercises

Set up user frontend

- if you want to take another step with this exercise you can create some kind of user frontend
- can use HTML, javascript, jquery/some javascript framework
- use the backend messages and status codes to determine the *frontend* routes the end user can access

Expand on the backend client data storage

- we could also try to connect our backend server to some kind of database
  - non-relational ones are easier (e.g. mongoDB)
- we can try to store multiple client usernames and passwords
  - be careful with how you store passwords
  - might want to research about packages for passwords
  - hashing, salting