Lab 2: Intro to NodeJS and ExpressJS

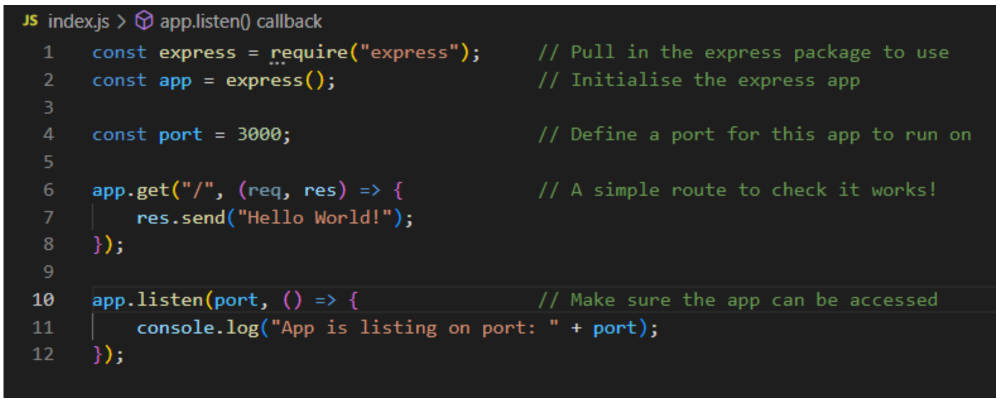
Learning Objectives

1. To learn how to create simple NodeJS and ExpressJS apps
2. Create your first web server and API
3. Create a fully functioning API

Exercise 1: Creating our first NodeJS application

Note: If you are on your own laptop, first make sure that you have installed NodeJS and VSCode

1. Create a new directory for your application and open the directory in your terminal. (The easiest way to do this is to open the directory in VSCode and click "Terminal" and then "New Terminal"
2. In the terminal, initialise a new NodeJS project by typing "npm init"
3. The terminal will ask you some questions about your project, feel free to answer any of the easy questions (e.g., your name, the projects name). For everything else, just click "enter" on your keyboard to use the default value.
4. On completion, you will have set the directory up as a Node application. There is no code yet, but you will have a package.json file. This file contains all your projects meta-data – including the answers to all the questions you have just answered (or skipped through)
5. Familiarise yourself with the contents of your package.json file. Note that the "main" script points to a file called "index.js" – this is the entry point to our application and the file that we first need to create to run code.
6. In the directory, create the "index.js" file.
7. NodeJS includes 1000's of libraries that can provide our applications with functionality without having to reinvent the wheel every time. For creating web applications we can use a popular library called ExpressJS. We first need to install this library into our project. In your terminal window, type "npm install --save express" – on completion, you should be able to see express added as a dependency to your package.json file.
8. Now that we have ExpressJS installed, open up the index.js file and let's write some code:

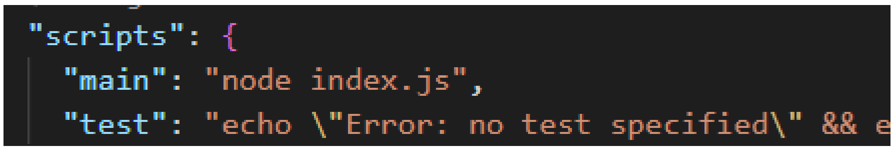


```
1   const express = require("express");      // Pull in the express package to use
2   const app = express();                    // Initialise the express app
3
4   const port = 3000;                        // Define a port for this app to run on
5
6   app.get("/", (req, res) => {              // A simple route to check it works!
7       res.send("Hello World!");
8   });
9
10  app.listen(port, () => {                  // Make sure the app can be accessed
11      console.log("App is listing on port: " + port);
12  });
```

9. It is important that you understand the code here, as it's the basis for writing all server applications in ExpressJS.
   a. On line 1, we import the ExpressJS library to use in our application
   b. On line 2, we initialise a new ExpressJS application
   c. On line 4, we set the port that our server will run on

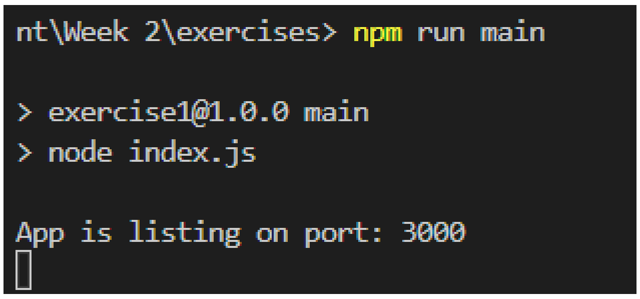
   d. On line 6, we create our first endpoint. The endpoint uses the HTTP GET method and takes in two parameters:
      i. The first is the route that the client needs to hit. In this case, a GET request to / (i.e., the root path of the server)
      ii. The second is a function that will trigger when the endpoint is hit by the client. The function takes in two parameters, one with all of the request information (which I've called req) and one with all of the response information (which I've called res).
   e. On line 7, you can see that when someone triggers the function (by sending a GET request to "/"), the server will simply send the text "Hello World" back to the client (using the response variable).
   f. On line 10, we set the server listening for requests. Again this function has two parameters, the first is the port that the application needs to listen on, and the second is a function that triggers once the server is listening and ready for requests.

10. Now that we have some code, add the following line to the "scripts" section of the package.json file. Instructing the application to run our index.js file when we run the server.



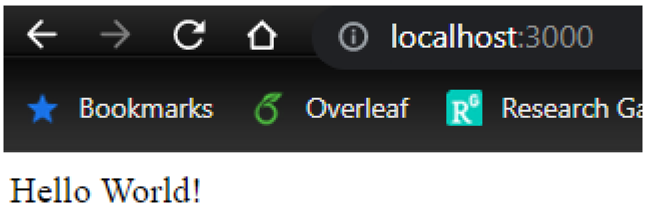
```
"scripts": {
  "main": "node index.js",
  "test": "echo \"Error: no test specified\" && e
```

11. In the terminal, run "npm run main"



```
nt\Week 2\exercises> npm run main

> exercise1@1.0.0 main
> node index.js

App is listing on port: 3000
```

12. Check that your app is working, both in your browser, and in Postman



Hello World!

Yay! First server! – But it doesn't do much…

Exercise 2: Adding some functionality

We want to create the following API:

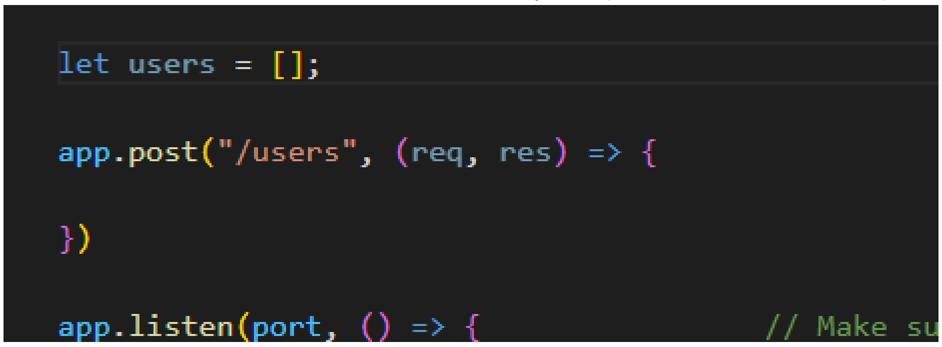
METHOD	ROUTE	DESCRIPTION
GET	/users	Gets a list of all users
POST	/users	Adds a new user
PATCH	/users/:id	Updates a user at a specific ID
DELETE	/users/:id	Deletes a user at a specific ID

Added complexity:

- New methods
- Handling path parameters (ID's in the routes)
- Handling data (both in the request, and in the response)

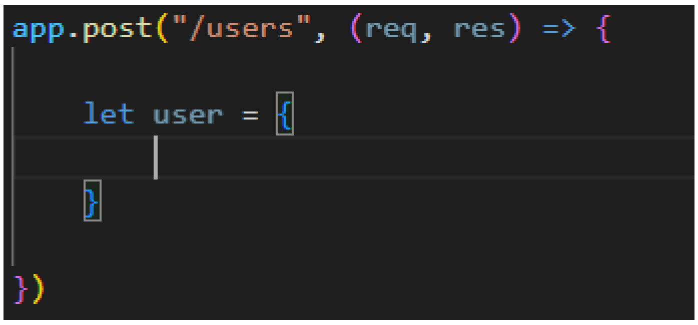
Exercise 2.1: POST /users

1. First we need to create somewhere to store our users – in this case, we can create an array (in later weeks, we will use a database for storing data)
2. We can also create the scaffold for the new endpoint (note the new method!)



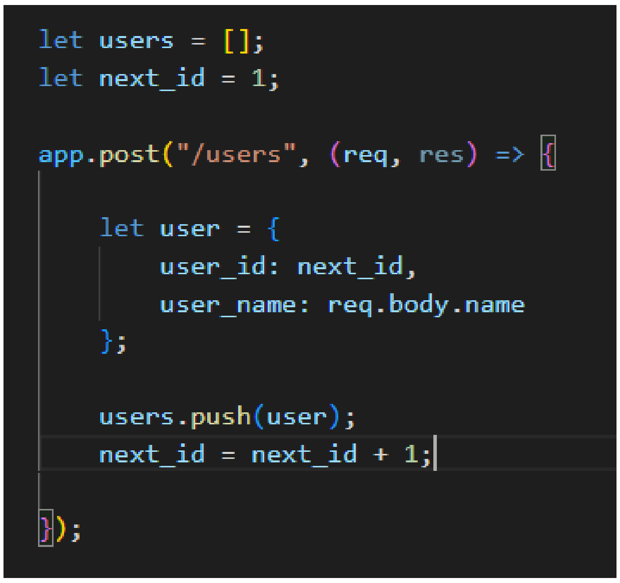
```
let users = [];

app.post("/users", (req, res) => {

})

app.listen(port, () => {                    // Make su
```

3. To start implementing the endpoint, start by creating the object that we want to add to our array.



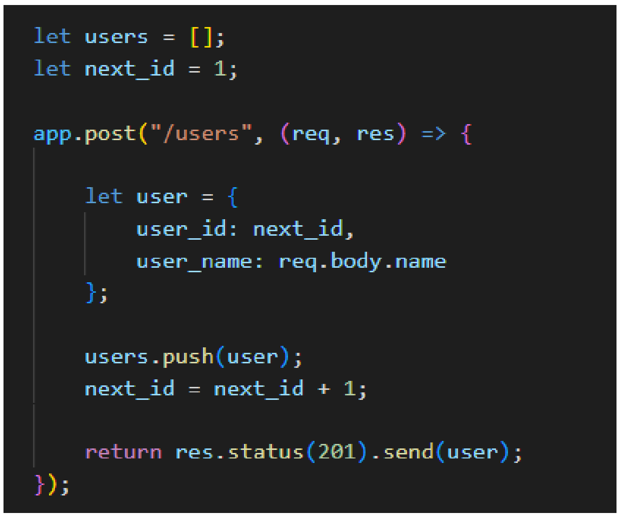
```
app.post("/users", (req, res) => {

    let user = {

    }

})
```

4. We can structure the object however we want (or however the specification says that we should). The data being sent is pulled out of the body of the request (stored in req.body). Look at the image below. Note the following:
   a. We add the object to the array using the JavaScript "push" method
   b. We have creating a "next_id" variable for automatically assigning ID's to the users



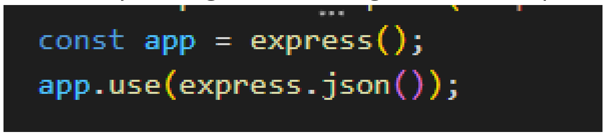
```
let users = [];
let next_id = 1;

app.post("/users", (req, res) => {

    let user = {
        user_id: next_id,
        user_name: req.body.name
    };

    users.push(user);
    next_id = next_id + 1;

});
```

5. We need to send some sort of a response to the client so that they know the server is finished processing their request. We do this below by sending the new user object back to the client along with a 201 Created HTTP status.



```
let users = [];
let next_id = 1;

app.post("/users", (req, res) => {

    let user = {
        user_id: next_id,
        user_name: req.body.name
    };

    users.push(user);
    next_id = next_id + 1;

    return res.status(201).send(user);
});
```

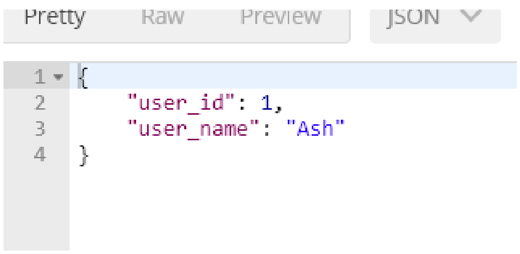
6. Finally, we need to tell the server that we are working with JSON data. We can do this easily for the entire application by adding the following line at the top of our file



```
const app = express();
app.use(express.json());
```

7. Save and restart your server. Test your endpoint by sending a POST request in Postman. Think about:
    a. What's the method you need to send?
    b. What's the route?
    c. What data do you need to send?

Note: to stop the server running, hit CTRL + C in the terminal window

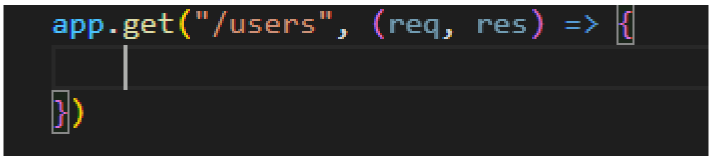


```
{
    "user_id": 1,
    "user_name": "Ash"
}
```

Exercise 2.2: GET /users

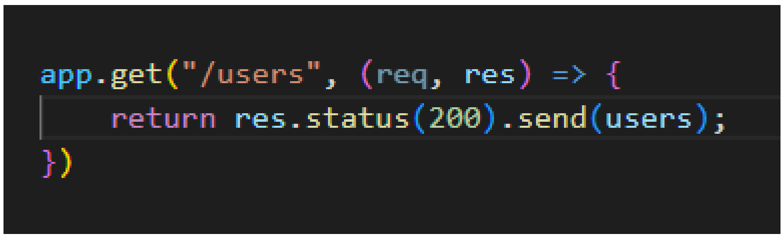
Now that we can add users, let's implement the GET endpoint so that we can view all the users in our application.

1. Start with the scaffold function as we did in the last exercise. This time the method will be a GET.



```
app.get("/users", (req, res) => {
    
})
```

2. This one is pretty simple – the array already contains a list of JSON objects in a suitable format, so we can just return it



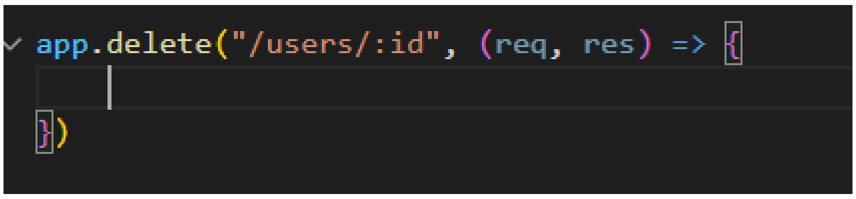
```
app.get("/users", (req, res) => {
    return res.status(200).send(users);
})
```

3. Save, restart the server, and test in Postman.
    a. **You have just created users when testing the last exercise – so why does this return an empty list? Create some users and test again**

Exercise 2.3: DELETE /users/:id

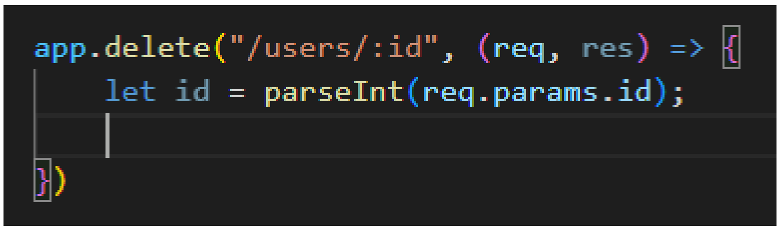
We can add and view our list of users, but what about deleting users? We need to specify a specific record in the array to operate on – we do this with by specifying the ID as a path parameter in the route.

1. Start with the endpoint scaffold, only using the delete method.



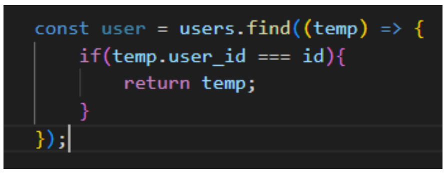
```
app.delete("/users/:id", (req, res) => {
    
})
```

2. We can get the path parameters from **req.params**. The route is a string, so we need to make sure to convert it to an integer with the parseInt() function.

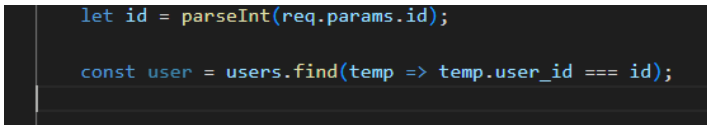


```
app.delete("/users/:id", (req, res) => {
    let id = parseInt(req.params.id);
    
})
```

3. We can search for a specific user using the JS **find()** function.
    a. The find() function takes in a function as a parameter.
    b. This will loop through the list of users. If the user_id is equal to our path parameter (in the ID variable) then we will return that user and store it in the user variable.
    c. **Note**: a triple equals (===) in JS means compare the value AND the datatype
    d. **This function can be written as a one line function instead**…

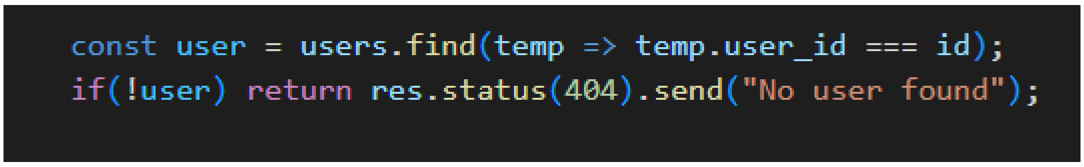


```
const user = users.find((temp) => {
    if(temp.user_id === id){
        return temp;
    }
});
```



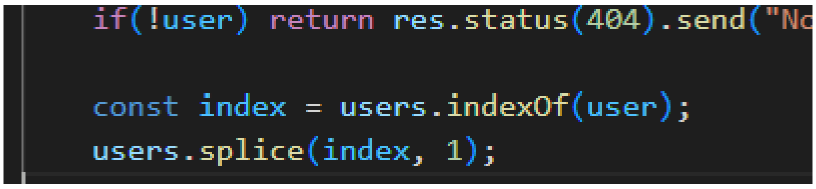
```
let id = parseInt(req.params.id);

const user = users.find(temp => temp.user_id === id);
```

4. If the find() function doesn't find a user with the sought after ID, we need to send a 404 Not Found response.



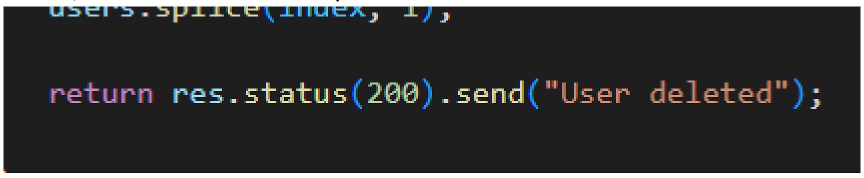
```
const user = users.find(temp => temp.user_id === id);
if(!user) return res.status(404).send("No user found");
```

5. If the user is found, we need to find the index of that user in the list. Then we can use the JS **splice()** function to remove it.
   a. **Note:** the splice function takes in two parameters. The first is the index of where to start splicing, and the second denotes how many items to remove (in this case, just one)



```
if(!user) return res.status(404).send("No

const index = users.indexOf(user);
users.splice(index, 1);
```

6. If all has worked, let's return a 200 response to the client



```
users.splice(index, 1);

return res.status(200).send("User deleted");
```

7. Save, restart, and test. What happens if the same user is deleted twice?

Exercise 2.4: PATCH /users/:id

Finally, we need to be able to update a user. You now know enough to do this without much guidance (you've already written most of the code)

1. Start with the normal endpoint scaffold, only using the PATCH method
2. Get the ID path parameter and return a 404 if it doesn't exist
3. If the name in the request body is different to the existing users name, then overwrite it
4. Send a 200 message to let the client know that the user has been updated
5. Save, restart, and test.

Exercise 3: Extend your API

Create an API endpoint to view a single user.

- What should the method be?
- What should the route be?
- What if we can't find the user in the list?
- What do we send back to the client?

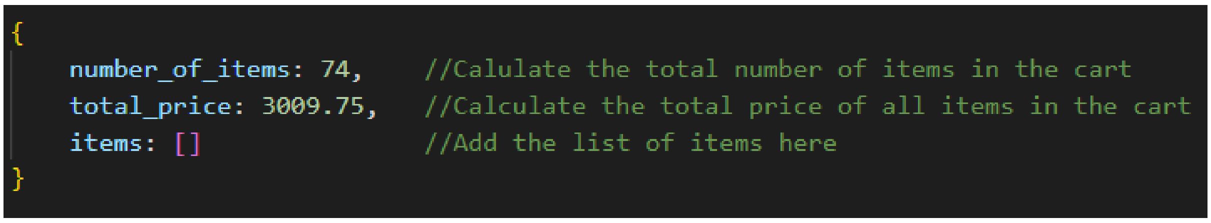
Exercise 4: ShoppingCartAPI

Using everything you have learnt, create a new application that adheres to the following API specification:

METHOD	ROUTE	REQUEST	RESPONSE	DESCRIPTION
GET	/cart	-	[cart_items]	Returns a list of all items in the cart
POST	/cart	item_name, item_price, quantity	item_id, item_name, item_price, quantity	Adds a new item to the cart
GET	/cart/:id	-	item_id, item_name, item_price, quantity	Gets a single item from the cart
PATCH	/cart/:id	quantity	item_id, item_name, item_price, quantity	Updates a single item within the list (users can only update the quantity of items)
DELETE	/cart/:id	-	"<Item name> Deleted"	Deletes an item from the list

Exercise 5: Become a JS master

Change your ShoppingCartAPI implementation so that when a client sends a GET request to /cart, the responding JSON is in the following formal:



```
{
    number_of_items: 74,    //Calulate the total number of items in the cart
    total_price: 3009.75,   //Calculate the total price of all items in the cart
    items: []               //Add the list of items here
}
```

Exercise 6: Get assignment ready!

Look at the assignment checklist items for this week. Familiarise yourself with the API you will need to implement and get your starter code downloaded and running. Can you run the tests?