JavaScript on the back end

Learning Objectives

- Understand what back-end/server side means
- Understand what a servers job is
- Understand the basics of data persistence
- Be able to write to/read from a .json file
- Be able to send data from an express app to the browser

Intro

So far most of our understanding was mostly technical/theoretical. What we were working with were examples made to understand and dissect complex ideas and concepts. The examples and analogies were very simplified to make sure that instead of worrying about the problem of complexity, we could focus on the concepts themselves.

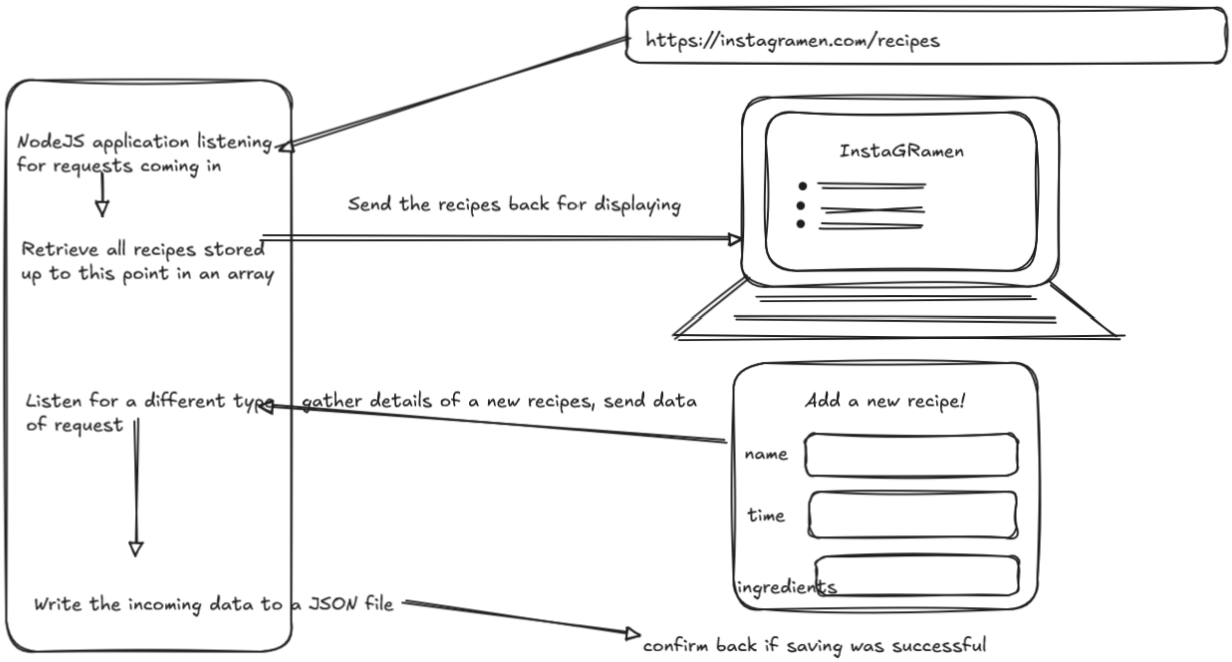
But the promise I made at the beginning of the course was about creating a full stack app. A real life, widescreen, technicolor app that is able to do more on the front end than just be a fancy static html/css page with no behaviour and dynamism whatsoever. And the back end is also more than just looping through an array, and logging values out to the console.

Let's consider the long-discussed example I kept coming back to: An app to display and store recipes.

The Back End of a recipe app

We probably have a rough idea of how the front end of a recipe app might look like or behave, and the reason for that is simple: we have seen examples, and interacted with them on a regular basis. So if I had to ask you, you'd be able to tell me what the behaviour is that is expected from a website that stores recipes for us, even if you had no immediate idea on how to execute the code needed to create a front end for our app.

The back end, however, is more of a mystery at first glance. It's not something most of us had to think about, much less plan and create it. So let's try to visualise the role and behaviour of the back end of our app!



Just to create or retrieve recipes, we only need a couple of dozens of lines of code. But to understand what that does, we need to dig a little deeper.

There are 4 concepts we need to talk about before we can get started:

1. NPM
2. Servers
3. HTTP
4. Data persistence (JSON)

Since the video only have a limited time to cover these concepts, please feel free to pause, reflect, return later, or ask to cover these concepts using an AI model!

Let's start with a very helpful tool: NPM!

NPM

Nowadays when a complex programming problem rears its ug... I mean beautiful head, I can guarantee you that it's not the first time a developer had to solve said problem (apart from some *real* cutting edge companies and products)

Most of the problems, while being unique, can be solved in a very uniform fashion.

For example, for a JavaScript back end, we need to create something called a *web server*. It's a common task to do, regardless of what your web application actually does.

Since it's a common problem, developers realised that reinventing the wheel every time something needs to be rolled around is painfully inefficient. And to make not just their, but their colleagues and fellow developers lives easier, they came up with the idea of libraries or packages. These packages serve as code written by other developers that are usually open source, meaning anyone can see the source code and can offer contributions.

It's really hard to find a real world analogy for this, since the concept is so alien for other professions - you don't see (often, at least) chefs offering up their revolutionary new recipes for free, or inventors puttin up patents for free!

One of the best package managers in the world is the NPM (Node Package Manager) registry. From there, millions of packages are available, with packages being downloaded by the billions each month!

Luckily, NodeJS came with NPM installed, since they are very well integrated. We will use NPM for our web server needs.

In order to start utilising the NPM package manager, we need the following steps:

0. (You should start VS Code as an administrator - this will give us admin privileges for the terminal we are about to open. Going forward this is the best approach!)
1. Create a folder for our session (this will be the app's home or root directory)
2. Right click on the folder's name, select `Open in Integrated Terminal`. This opens up the terminal of your operating system at the same location as your folder.
3. Copy or type in the following command: `npm init -y`

This will create a file called `package.json`. We will not cover too much in depth what this is doing under the hood - imagine this as an ID card for our project, detailing everything worth noting about it.

Before we start using NPM, let's jump ahead a little bit to our 4th point - JSON.

JSON

JSON (short for JavaScript Object Notation) is basically a way to store and share data in a simple, readable format. It's like a text-based "note" that both humans and computers can easily understand. Its other strong benefit can be extrapolated from its name: It is really easy to turn it to JavaScript objects (since they are just a bunch of key-value pairs), and vice versa - JS objects can be easily turned into JSON data.

Here are some benefits:

1. It Looks Like JavaScript Objects:

It's just a bunch of key: value pairs wrapped in curly braces {}. For example:

```
{
  "name": "John",
  "age": 30,
  "isCool": true
}
```

> Note: Unlike regular JavaScript objects, JSON data needs to have all its keys in double qoutes `""`

2. It's Flexible:

You can use it for strings ("hello"), numbers (42), booleans (true/false), arrays ([1, 2, 3]), other objects ({ "key": "value" }), and even null.

3. Everyone (and everything) Uses It:

It's the go-to format for APIs (how apps talk to each other), config files, and storing data. It's not just for JavaScript - it works with almost every programming language.

4. It's Lightweight:

No extra fluss, just the data you need, making it fast to send over the internet. It can be turned into pure text, making it simple to send and receive.

In the case of NPM, we use JSON as a configuration format, but later we will use it for both data persistence and data delivery via the request-response cycle.

Adding our first NPM package

Now that we know (kinda) what JSON is good for, let's add something to our JSON. We could do it manually, but the chance for adding a typo and messing it up is a bit too high to my liking. Alternatively, we could use the terminal, and enter other npm commands there to install packages.

Copy or enter the following into your terminal window: `npm install express`

(remember, run VS code as an admin, and make sure you run the command in the terminal! Follow the video if in doubt)

Once installed we actually can see the fruit of our (or npm's?) labour - it created a `node_modules` folder!

To cut a long story short, this folder holds all our important downloaded packages, essentially other people's code. Think of NPM as a whole like a hardware store, and our node_modules as our mobile toolbox. We can get any tools we want using `package.json` (our shipping list, if you will!), and it will go and grab it for us - for free! If we want a new project we just need to add the items using `npm` commands in terminal or the package.json file.

Now that we know about NPM, it's time to move on to create our server via Express.

But wait... what is a server...?

(Web) Servers

A web server is a software or hardware system that processes incoming HTTP requests from clients (e.g., web browsers, mobile apps) and sends back HTTP responses (e.g., HTML pages, JSON data, images, etc.). It acts as the backbone of the internet, enabling communication between clients and servers.

Key Functions of a Web Server:

- Receives requests from clients using the HTTP (Hypertext Transfer Protocol) or HTTPS (secure HTTP) protocol. It does this using certain HTTP methods, like `GET`, `POST` or `DELETE`.
- Serves Static and Dynamic Content.
    - Static Content: Files like HTML, CSS, JavaScript, images, and videos that don't change (e.g., a blog post).

    - Dynamic Content: Data generated on-the-fly based on user input or other factors (e.g., a personalized dashboard).

- Manages Routing: Determines how to handle requests based on the URL (e.g., /home, /about, /api/data).
- Processes Backend Logic: Executes server-side code (e.g., querying a database, performing calculations) to generate responses.
- Sends HTTP Responses: Returns data to the client, often in the form of HTML, JSON, or other formats. Includes status codes (e.g., 200 OK, 404 Not Found, 500 Internal Server Error) to indicate the result of the request.

Components of a Web Server:

1. Hardware: A physical computer or server that hosts the web server software and stores website files.
2. Software: The program that handles HTTP requests and responses. Examples include:
    - Apache: A widely used open-source web server.
    - Nginx: Known for high performance and scalability.
    - IIS: Microsoft's web server for Windows.
    - Node.js: A runtime environment that can act as a web server using frameworks like Express.
3. Database: Stores and retrieves data for dynamic content (e.g., user accounts, product information).
4. Middleware: Software that sits between the web server and the application, handling tasks like authentication, logging, and data parsing.

Now that we know what are web servers, and (roughly) how they work, we must also talk a bit more about what we covered at the very first lesson!

HTTP

HTTP (Hypertext Transfer Protocol) is the "language" that your web browser and websites use to talk to each other. It's the foundation of how the internet works - every time you visit a website, send a form, or load an image, HTTP is behind the scenes making it happen.

If we break it down a bit further, `Hypertext` is probably the only word that sounds alien a bit, but if you joined last year's Couch to Coder or if you ever played around with HTML, you probably heard about it - `HTML` stands for `HyperText Markup Language`, so `HTTP` is just the protocol used for transferring HTML files!

How HTTP Works:

1. **You Make a Request**:
    - When you type a URL (like `https://example.com`) into your browser, your browser sends an **HTTP request** to the server that hosts the website.
2. **The Server Responds**:

- The server receives your request, figures out what you're asking for (e.g., a webpage, an image, or data), and sends back an **HTTP response** with the requested content. Note that one single request could result in a lot of actions (e.g. a login request could log you in, find your profile data, find all the relevant social media news for you and also start downloading images for you)

3. **Your Browser Displays the Content**:
    - Your browser takes the response (like HTML, images, or JSON) and shows it to you as a webpage or app. This is the job of the front end, or client side of our application.

(Optional) Key Parts of HTTP:

You are absolutely fine to disregard the next section, but if you're interested in the finer details of the request/response cycle, this could prove useful!

When an HTTP request is being made, the following parts are important to know about

1. **Methods (Verbs)**:
    - These tell the server what kind of action you want to perform. Think of the URL you enter as the phone number, and the verb is the extension that will tell the back end exactly which person to connect you to! The most common ones are:
        - `GET`: Fetch something (e.g., load a webpage).
        - `POST`: Send data (e.g., submit a form).
        - `PUT`: Update something.
        - `DELETE`: Remove something.
2. **Status Codes**:
    - These are like "short messages" from the server to tell you what happened. For example:
        - `200 OK`: Everything worked!
        - `404 Not Found`: The page doesn't exist.
        - `500 Internal Server Error`: Something went wrong on the server.

        > Note: These are agreed upon by the web development community and its up to the devs to use them appropriately. You can absolutely send back a 200 code even if the user is requesting a nonexistent page - but please don't do that!
3. **Headers**:
    - These are extra bits of information sent with requests and responses. For example:
        - Headers can tell the server what type of data your browser can handle (`Accept: application/json`).
        - They can also include cookies for tracking or authentication.
4. **Body**:
    - This is the actual data being sent (e.g., form data in a `POST` request or the HTML of a webpage in a response).

Example of an HTTP Request and Response:

1. **Request**:

```
GET /about HTTP/1.1
Host: example.com
```

   - This means: "Hey, `example.com`, can I get the `/about` page?"

2. **Response**:

```
HTTP/1.1 200 OK
Content-Type: text/html

<html>
  <body>
    <h1>About Us</h1>
  </body>
</html>
```

   - This means: "Sure! Here's the HTML for the `/about` page."

Why HTTP Matters:

- **It's Everywhere**: (Almost) every website, API, and app uses HTTP to communicate.
- **It's Simple**: It's easy for both humans and machines to understand.
- **It's Flexible**: It works for everything from loading a webpage to sending data to a mobile app.

Data persistence

So far so good. The last piece of the puzzle is storing our data.

Last time we've seen how we can manipulate arrays, add, remove, modify items. The fact is, however, is that as soon as we restart our script, everything gets set back to square one. No changes remained, deleted items magically reappearing, added items sent to the void.

The reason for this is pretty straightforward - we never told our code to persist these changes, we just wrote step by step instructions to modify an array or object. To persist these modifications, we need to change the way we think about our code.

When we are working with a spreadsheet, the data is stored within the file we create and modify, even if it's not explained to us in detail - Excel/Sheets are just tools to modify that data, somewhere all our values are being stored!

There are many ways to persist data for web programming - the most common one is to use a database engine, like a flavour of SQL (Postgres/SQLite/etc), or a noSQL tool like MongoDB.

However, we are quite time restricted, therefore I offer a different solution: Since we are already a bit familiar with JSON, we could use this to our advantage. If you remember, an array is a perfectly fine datatype to store in a JSON file, and what is a collection of recipes if not one big array with plenty of recipe objects stored in it?

Reading and writing from/to JSON files have certain drawbacks:

- Not really scalable, if we get thousands of requests, there is a good chance of either missing data, or slowing our app down to a halt
- Not structured well enough, we could store random values that do not look like recipes at all, and cause bugs or errors in our app
- we cannot connect to different files - we need to set one file to read/write from, which has limited capacity

However, it is not unheard of: Some settings, configurations or bacis data can sometimes be stored in JSON files, and even if it is not the best solution, we can always refactor in the future - if the rest of our logic is sound, it doesn't matter what tool is being used to save our data on our machine!

Our code will simply read in data from a `recipes.json` file, from which the data can be easily turned into a JavaScript array of objects!

Now that we know what we are going for, let's write the little code we need. Let's create our file - and out of convention, let's call it `index.js` in our session_3 folder!

First, let's type (or copy from the notes) the following code:

```
const express = require('express');
const app = express();
const port = 3000;
```

There are a lot going on under the hood, so I'll keep it brief for now:

- The first line of code imports the downloaded package `express` and stores it in a variable to use later.
- The second line calls a function that gives us back a big and complex JavaScript object that can act as our web server. There is plenty more going on, but for now, let's leave it at that.
- The third line creates a variable that will store the port number of our web server. Port numbers act like individual channels that helps computers communicate over a network. It would make servers and computers pretty one sided and relatively cumbersome if we only could only expose a single thing per IP address - luckily, each computer has thousands of ports that can be used to run tools and apps locally, or to expose these channels to a network - like the internet - to be connected to. Out of convention our chosen express port number will be 3000!

Next we need to create the code that will handle the incoming requests from the front end of our app. For this course, we will only need 2 endpoints - One for a home page, one to send back all recipes, and one to store an incoming recipe.

```
app.get('/', (req, res) => {
    res.send('Hello, World! Home page here!');
});
app.get('/recipes', (req, res) => {
    res.send('Hello, World! Sending back all the recipes!');
});
app.post('/recipes', (req, res) => {
```

```
    res.send('Hello, World! Recipe accepted, we are storing your favourite
dishes');
});
```

So what's happening here?

As we discussed before, `GET` and `POST` are agreed upon verbs to use when we are trying to retrieve from/send to a server data. Express has methods that someone built in for us in order to use, which makes our life considerably easier!

The weird brackets and the arrow are syntax rules for functions - every programming language has them, but only a couple has this weird concept of adding a function inside another function call. We won't have time to cover the way they are being used now - we will touch upon them during the data analytics advanced extra session, but if you are extra curious - nothing is stopping you from asking your studdybuddy the question of what are functions, and why do we like them?

We also have `"/recipes"` in the get/post methods. This is going to be part of the urls we can use, since each path needs to be unique. This will make our paths to be `localhost:3000/recipes`

> Note: even though both paths are `/recipes`, we use different verbs, one is "GET localhost:3000/recipes", the other is "POST localhost:3000/recipes". Think of the phone extension analogy - both are the same phone number, but they connect you to different people!

Finally, we have the `req`, `res`, and `res.send()` - `req` and `res` are just 2 variables we can name whatever we want, but, again, we follow convention.

> Note: The fact that we are using `get()` or `post()` doesn't inherently do much - we could retrieve data with a `post` or even a `delete` method, and create things with a `get`. It's more like an agreement in the web development community and drifting away from it could cause a lot of headaches if someone more experienced were to encounter your code!

We are not ready to test our app yet - we need one more step.

The code above only created the routes we can interact with from our client side app. We still need to tell our code that we would like to start our server, and it should start listening to potential incoming requests.

Add the following:

```
app.listen(port, () => {
    console.log("Server is running on http://localhost: ", port);
});
```

After clicking on the play button, we can see in our output in VS Code the message added in our console.log, which means that as long as we keep this app running, we will be able to interact with our server.

Let's test if everything is working alright! Open up a Chrome browser, and type in the following in the `url` bar: `localhost:3000/recipes` or `localhost:3000`, and lo and behold! We have our messages being displayed on the screen!

However, until we do not have JavaScript code on our front end (we don't even have a front end yet!), unfortunately we do not have an easy way to access the `post` route. For now, we can type the following into our terminal:

```
curl -X POST http://localhost:3000/recipes
```

This will give us back the other message, from our creating route. Once we get out front end, this will be way smoother!

Now we just need to make sure we are sending back JSON data instead of just a message!

Let's try creating a JavaScript array first with some objects in it.

Copy and paste the following code under the port number, but above the `app.get()` line:

```
const recipes = [
    {
        name: "Spaghetti Carbonara",
        time: "30 minutes",
        cuisine: "Italian",
        ingredients: [
            "200g spaghetti",
            "2 large eggs",
            "100g grated Parmesan cheese",
            "150g bacon, chopped",
            "2 cloves garlic, minced",
            "1/2 teaspoon black pepper",
            "Salt to taste"
        ],
        steps: [
            "Cook spaghetti according to package instructions.",
            "In a bowl, whisk eggs and grated Parmesan cheese.",
            "Cook bacon in a pan until crispy, then add minced garlic.",
            "Drain spaghetti and add it to the pan with bacon and garlic.",
            "Remove from heat, add the egg and cheese mixture, and toss quickly to
coat the spaghetti.",
            "Season with black pepper and salt to taste.",
            "Serve immediately."
        ]
    },
    {
        name: "Chicken Tikka Masala",
        time: "1 hour",
        cuisine: "Indian",
        ingredients: [
            "500g chicken breast, cut into pieces",
            "200g yogurt",
            "400g tomato puree",
            "1 large onion, finely chopped",
            "3 cloves garlic, minced",
            "1 tablespoon ginger, minced",
            "1 tablespoon garam masala",
            "1 teaspoon turmeric",
```

```
            "1 teaspoon cumin",
            "1 teaspoon coriander",
            "100ml cream",
            "2 tablespoons oil",
            "Salt to taste"
        ],
        steps: [
            "Marinate chicken in yogurt, garlic, ginger, and spices for 30
minutes.",
            "Heat oil in a pan and cook the marinated chicken until browned.",
            "Remove chicken and sauté chopped onions in the same pan.",
            "Add tomato puree, spices, and cook until the mixture thickens.",
            "Add the chicken back to the pan and simmer for 15 minutes.",
            "Stir in cream and cook for another 5 minutes.",
            "Serve with rice or naan."
        ]
    },
    {
        name: "Chocolate Chip Cookies",
        time: "25 minutes",
        cuisine: "American",
        ingredients: [
            "250g flour",
            "200g butter, softened",
            "150g sugar",
            "150g brown sugar",
            "2 large eggs",
            "1 teaspoon vanilla extract",
            "1 teaspoon baking soda",
            "1/2 teaspoon salt",
            "200g chocolate chips"
        ],
        steps: [
            "Preheat oven to 375°F (190°C).",
            "In a bowl, cream together butter, sugar, and brown sugar.",
            "Beat in eggs and vanilla extract.",
            "Mix in flour, baking soda, and salt.",
            "Fold in chocolate chips.",
            "Drop spoonfuls of dough onto a baking sheet.",
            "Bake for 10-12 minutes or until golden brown.",
            "Let cool before serving."
        ]
    }
];
```

And now let's change our `get` route to return this data instead of the simple string message:

```
app.get('/recipes', (req, res) => {
    res.json(recipes);
});
```

!IMPORTANT! we need to restart our server every time we make changes in our back end code, otherwise the changes will not be applied.

Hit the `localhost:3000/recipes` endpoint again, and rejoice.

We could add more to our response if we wanted to - HTTP code, extra messages, authentication tokens, but for now, we should leave it at that.

The next step is to move our recipes to a .json file, which will act like our database. Create a file right next to our index.js called `recipes.json`, and copy the following code in there (it is very similar to our recipes above, but JSON can't handle variable assignment or keys without double quotes, so it is slightly modified to accomodate to these changes.) Don't forget to save the file!

```
[
    {
        "name": "Spaghetti Carbonara",
        "time": "30 minutes",
        "cuisine": "Italian",
        "ingredients": [
            "200g spaghetti",
            "2 large eggs",
            "100g grated Parmesan cheese",
            "150g bacon, chopped",
            "2 cloves garlic, minced",
            "1/2 teaspoon black pepper",
            "Salt to taste"
        ],
        "steps": [
            "Cook spaghetti according to package instructions.",
            "In a bowl, whisk eggs and grated Parmesan cheese.",
            "Cook bacon in a pan until crispy, then add minced garlic.",
            "Drain spaghetti and add it to the pan with bacon and garlic.",
            "Remove from heat, add the egg and cheese mixture, and toss quickly to
coat the spaghetti.",
            "Season with black pepper and salt to taste.",
            "Serve immediately."
        ]
    },
    {
        "name": "Chicken Tikka Masala",
        "time": "1 hour",
        "cuisine": "Indian",
        "ingredients": [
            "500g chicken breast, cut into pieces",
            "200g yogurt",
            "400g tomato puree",
            "1 large onion, finely chopped",
            "3 cloves garlic, minced",
            "1 tablespoon ginger, minced",
            "1 tablespoon garam masala",
            "1 teaspoon turmeric",
            "1 teaspoon cumin",
```

```
            "1 teaspoon coriander",
            "100ml cream",
            "2 tablespoons oil",
            "Salt to taste"
        ],
        "steps": [
            "Marinate chicken in yogurt, garlic, ginger, and spices for 30
minutes.",
            "Heat oil in a pan and cook the marinated chicken until browned.",
            "Remove chicken and sauté chopped onions in the same pan.",
            "Add tomato puree, spices, and cook until the mixture thickens.",
            "Add the chicken back to the pan and simmer for 15 minutes.",
            "Stir in cream and cook for another 5 minutes.",
            "Serve with rice or naan."
        ]
    },
    {
        "name": "Chocolate Chip Cookies",
        "time": "25 minutes",
        "cuisine": "American",
        "ingredients": [
            "250g flour",
            "200g butter, softened",
            "150g sugar",
            "150g brown sugar",
            "2 large eggs",
            "1 teaspoon vanilla extract",
            "1 teaspoon baking soda",
            "1/2 teaspoon salt",
            "200g chocolate chips"
        ],
        "steps": [
            "Preheat oven to 375°F (190°C).",
            "In a bowl, cream together butter, sugar, and brown sugar.",
            "Beat in eggs and vanilla extract.",
            "Mix in flour, baking soda, and salt.",
            "Fold in chocolate chips.",
            "Drop spoonfuls of dough onto a baking sheet.",
            "Bake for 10-12 minutes or until golden brown.",
            "Let cool before serving."
        ]
    }
]
```

Then we will need some slight modifications in our `index.js` file. First, we will need a built in package to handle file reading and finding the path to our file. Built in packages are the same as NPM packages, but they do not need to be installed!

```
const express = require('express');
const fs = require('fs'); // added
const path = require('path'); // added
```

```
const app = express();
const port = 3000;
```

Then we use our code to find the path to our json file - this is a bit lengthy, but helps us in the future, so regardless of how we run our app, the file will be found!

```
const port = 3000;

const recipesFilePath = path.join(__dirname, 'recipes.json');
```

Finally we will modify our `get` route for our recipes:

```
app.get('/recipes', (req, res) => {
// Read the JSON file
  fs.readFile(recipesFilePath, 'utf8', (err, data) => {
    // we read the file located at the filepath, encoding is utf8, and, just like req/res, we have an err/data variable
    // if we have something in the err variable, we could handle that with an if statement, for now, let's keep it simple
    // data will hold the data read in from the file

    // Parse the JSON data from the file, turning it into JSON data.
    const recipes = JSON.parse(data);

    // Send the recipes as JSON response
    res.json(recipes);
  });
});
```

> Note: Please make sure you deleted the original array from index.js, we will not need it anymore!

Fantastic! Our data comes from a .json file that is separate from our code, and (once we have a front end) we can finally modify it. This will make sure that any changes we make (like adding a recipe) will be persistent, and even if we restart our app later, changes will stay with us!

The saving of new recipes will be a bit harder to do, but we will get it done as soon as we finish with next week's lesson, where we can create a nice form to add new recipes!