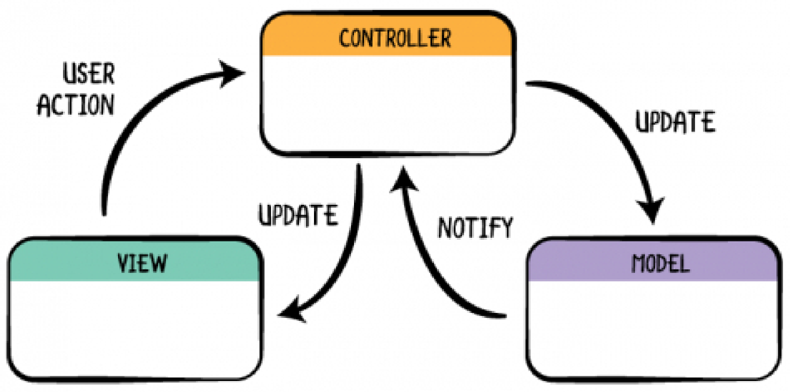
JSON and Asynchronous JS

MVC - Model View Controller

Model-View-Controller (MVC) is a popular design pattern used in software engineering to separate the data (model), the user interface (view), and the logic that controls them (controller).
Imagine you're building a website to display a list of books. The data (model) would be the list of books, the user interface (view) would be the HTML page that displays the books, and the logic (controller) would be the code that retrieves the list of books from the database and updates the view accordingly.
By using the MVC pattern, you can keep each aspect of the application separate, making it easier to maintain and scale the application as needed. For example, you could easily change the view to display the books in a different format without affecting the data or the logic.



In the context of Node.js development, the MVC pattern can be implemented using various frameworks and libraries such as Express.js, which is a popular web framework for Node.js.
Here's an example of how you might implement the MVC pattern in a Node.js application using Express.js:

1. Model: You would create a model to represent the data (books) you want to display on the website. This might be a JavaScript class with methods to fetch the books from the database.
2. View: You would create a view template in HTML to display the books. This template would be populated with the data from the model.
3. Controller: You would create a controller to handle the logic of fetching the data from the model and updating the view. In Express.js, this would typically be a route handler that uses the model to fetch the data and then render the view template with the data.

When a user visits the website, the controller would retrieve the data from the model, pass it to the view, and then render the HTML page. The user would see the list of books on the website, but they wouldn't have to worry about the data or the logic behind it.

By using the MVC pattern, you can create a clear separation of concerns in your Node.js application, making it easier to develop, maintain, and scale over time.

Introduction to Express.js

Express.js is a web framework for Node.js, a popular platform for building server-side applications using JavaScript. Express makes it easy to create web applications by providing a set of tools and libraries to handle common tasks, such as routing, middleware, and handling requests and responses.

Think of Express as a toolbox for building web applications. It provides a set of tools and components that you can use to build your application, just like you would use a hammer, saw, and screws to build a piece of furniture. With Express, you can create simple web applications that can handle HTTP requests, route traffic, and render web pages, among other things.

Express is a back end web application framework for building RESTful APIs with Node.js. It is designed for building web applications and APIs.

To install ExpressJS run the following command :

```
npm install express
```

Here's a simple example to help illustrate what Express can do:

Let's say you want to build a website that displays a "Hello World" message when someone visits the site. With Express, you can write a few lines of code to create a simple "Hello World" server, like this:

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

app.get("/", (req, res) => {
  res.send("Hello World");
});
```

```
app.listen(port, () => {
  console.log("Server is listening on port : ${port}");
});
```

In this example, you import the Express library and create an Express application. You then define a route that responds to GET requests to the root of the website ("/") by sending a "Hello World" message. Finally, you start the server and listen on port 3000, so that it's accessible from a web browser.

In conclusion, Express.js is a powerful and flexible framework for building web applications using Node.js. By using Express, you can write clean, organized, and maintainable code that is easy to understand and scale.

Routing with Express

Routing is the process of handling incoming HTTP requests and mapping them to specific actions in your web application. In Express, routing is achieved by defining "routes" that match specific URL patterns and responding with appropriate content.

Here's an example to help illustrate the concept of routing in Express:
Let's say you want to build a simple website that displays different content for different pages. For example, you might have a homepage at / that displays a welcome message, a "About" page at /about that gives an overview of the website, and a "Contact" page at /contact that shows a contact form.
With Express, you can write the following code to handle these different pages:

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

app.get("/", (req, res) => {
  res.send("Welcome to my website!");
});
```

```

app.get("/about", (req, res) => {
  res.send("This is the About page.");
});

app.get("/contact", (req, res) => {
  res.send("This is the Contact page.");
});

app.listen(port, () => {
  console.log("Server is listening on port ${port}");
});
```

In this example, you define three different routes that match the GET requests to the root ("`/`"), "`/about`", and "`/contact`" URLs. For each route, you define a callback function that returns the appropriate response when the route is matched.

When a user visits your website, the browser sends a GET request to the server. Express matches the request URL to one of the defined routes and returns the corresponding content. For example, if the user visits "`/`", Express will match the request to the first route and send the `"Welcome to my website!"` message.

In conclusion, routing is a fundamental part of building web applications with Express. By defining routes and handling incoming requests, you can create a flexible and scalable application that can handle different types of content and interactions with users.

There are 4 most commonly used HTTP methods GET, POST, PUT, and DELETE we have seen how to make GET requests lets see how other request are made

```
// GET request
app.get('/', (req, res) => {
  res.send('Hello, this is a GET request');
});

// POST request
app.post('/', (req, res) => {
  res.send('Hello, this is a POST request');
});

// PUT request
app.put('/', (req, res) => {
  res.send('Hello, this is a PUT request');
});

// DELETE request
app.delete('/', (req, res) => {
  res.send('Hello, this is a DELETE request');
});
```

In the example above, we're using `app.get()` to handle a GET request, `app.post()` to handle a POST request, `app.put()` to handle a PUT request, and `app.delete()` to handle a DELETE request. Each of these methods takes two parameters: the URL to match and a callback function that specifies what to do when a request to that URL is made.

When you run this code, you can send HTTP requests to `http://localhost:3000` using a tool like curl or a browser. If you send a GET request to `http://localhost:3000`, you should see the message Hello, this is a GET request in the response. Similarly, if you send a POST request, a PUT request, or a DELETE request, you should see the corresponding message in the response.

Serving static files in Express

Express.js provides a way to serve static files such as images, stylesheets, and JavaScript files. This means that you can store these files on your server and make them available to clients who make HTTP requests to your server.

Here's an example of how to serve static files using Express.js:

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

app.use(express.static('public'));

app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

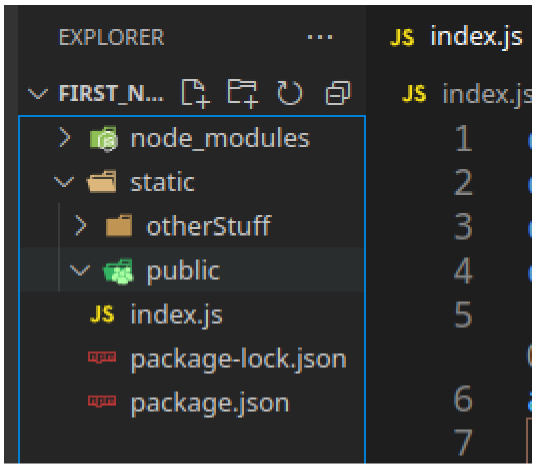
In this example, we're using the express.static() middleware function(we will learn about them later in the course) to serve all files in a folder called public as static files. This means that when a client makes an HTTP request for a file in the public folder, Express.js will serve that file to the client.

For example, if you have an image file called logo.png in the public folder, you can make an HTTP request for that file using a browser by visiting http://localhost:3000/logo.png. The browser should display the image in the response.

Note that the express.static() middleware function must be placed before any other routes in your Express.js application, because it will handle all requests that match files in the public folder, and you don't want other routes to interfere with that.

We can also create a folder named static and place all contents which are not going to be changed like images, css, we can also add a public folder inside it which will contain all the public files.

At this point the folder structure in VSCode looks like :


EXPLORER
FIRST_N...
node_modules
static
otherStuff
public
index.js
package-lock.json
package.json
index.js