Express JS

Now I mentioned that the framework that uses JavaScript is node, but strictly speaking, Node is actually not a framework. Instead it's a runtime environment.
Now we mentioned previously that what a runtime environment means is it allows us to run JavaScript on the computer. So Node is what enables us to use JavaScript on a computer and not just limited to within the browser.

Now express, on the other hand, is actually a JavaScript framework that allows us to create backends for our websites and with the powers of Node and Express combined is how most professional developers build their website backends.

Why Express JS?

Express.js is a small framework that works on top of Node.js web server functionality to simplify its APIs and add helpful new features. It makes it easier to organize your application's functionality with middleware and routing. It adds helpful utilities to Node.js HTTP objects and facilitates the rendering of dynamic HTTP objects.

Creating Our First Server with Express:

The server is just simply any computer and in most cases a server is a big and powerful computer that's on 24/7 and it's always listening for any requests, looking for a particular resource, like an HTML file of a website or the CSS or the JavaScript, etc.
Now, in addition to the server, the back end also consists of an application that is written using some sort of code, and that application is running on the server computer.
This could just be an index.js file that is on this server computer.
In addition, usually the back end also has a database, but this is usually in the case where you have a more complex back end, a more complex website where you have maybe user data or company data, etcetera.

- There are six steps to creating an express server. Which are shown below in fig.

Creating an Express Server

1. Create directory.
2. Create index.js file.
3. Initialise NPM.
4. Install the Express package.
5. Write Server application in index.js.
6. Start server.

1. **Create dir : > mkdir folder_name**

2. **Create index.js file : > New-Item -Path “index.js" -ItemType "File"**

3. **Initialise NPM : > npm init**

4. **Install Express : > npm i express**

5. **Writing Server App :**

```
import express from "express";
const app = express();
// create an instance of an Express application by calling the express function
const port = 3000;
app.listen(port, () => { console.log(`Server running on port ${port}.`); });

// The app.listen method tells the Express application to start a server and listen on the specified port (3000 in this case). The second argument is a callback function that runs when the server starts successfully. In this callback, a message is logged to the console indicating that the server is running and specifying the port number.
```

6. **Start Server : > node index.js**

// Now on web browser enter localhost:3000 and it’ll show meesage **“Can’t Get”.** It means it can't get our index page, our home page.

We can also check which ports on our computer are currently listening for interactions from the outside, by using this command:

> Netstat –ano | findstr “LISTENING”

HTTP Requests

HTTP stands for Hyper Text Transfer Protocol, the Transfer Protocol basically describes this as a language, a language that allows computers to talk to each other across the Internet.
When we make these HTTP requests, there are five main words that you'll come across and it's **GET, POST, PUT, PATCH and DELETE**. So let's go through each of them :

1. GET :
- Purpose: Retrieve data from the server.
- Usage: Used to request data from a specified resource.
- Idempotent: Yes (making multiple identical requests will result in the same response and state on the server).
- Example: Fetching user data.
  > app.get('/users', (req, res) => { // Code to retrieve and return users });

2. POST :
- Purpose: Send data to the server to create a new resource.
- Usage: Used to submit data to be processed to a specified resource.
- Idempotent: No (making multiple identical requests will create multiple resources). Example: Creating a new user.
- > app.post('/users', (req, res) => { // Code to create a new user });

3. PUT :
- Purpose: Update an existing resource or create a new resource if it does not exist.
- Usage: Used to send data to update/replace an existing resource.
- Idempotent: Yes (repeatedly updating the same resource with the same data will result in the same state).
- Example: Updating user data.
  > app.put('/users/:id', (req, res) => { // Code to update a user with the specified ID });

4. PATCH :
- Purpose: Apply partial modifications to a resource.
- Usage: Used to update a resource with partial data.
- Idempotent: Yes (repeatedly applying the same patch will result in the same state).
- Example: Partially updating user data.
  >app.patch('/users/:id', (req, res) => { // Code to partially update a user with the specified ID });

5. DELETE :
- Purpose: Delete a resource from the server.
- Usage: Used to remove a specified resource.
- Idempotent: Yes (deleting a resource multiple times will have the same result - the resource will no longer exist).
- Example: Deleting a user.
  > app.delete('/users/:id', (req, res) => { // Code to delete a user with the specified ID });

❖ **Now to solve previous problem of "Cannot Get/", we use get method to load our .html file :**

```
app.get("/", (req, res) => {
        res.send("<h1>Hello World!</h1>");
});
```

Route Definition: app.get("/") sets up a route to handle GET requests to the root path ("/").
Callback Function: (req, res) => { ... } defines what to do when this route is accessed. Sending **Response:** res.send("<h1>Hello World!</h1>") sends an HTML response back to the client.

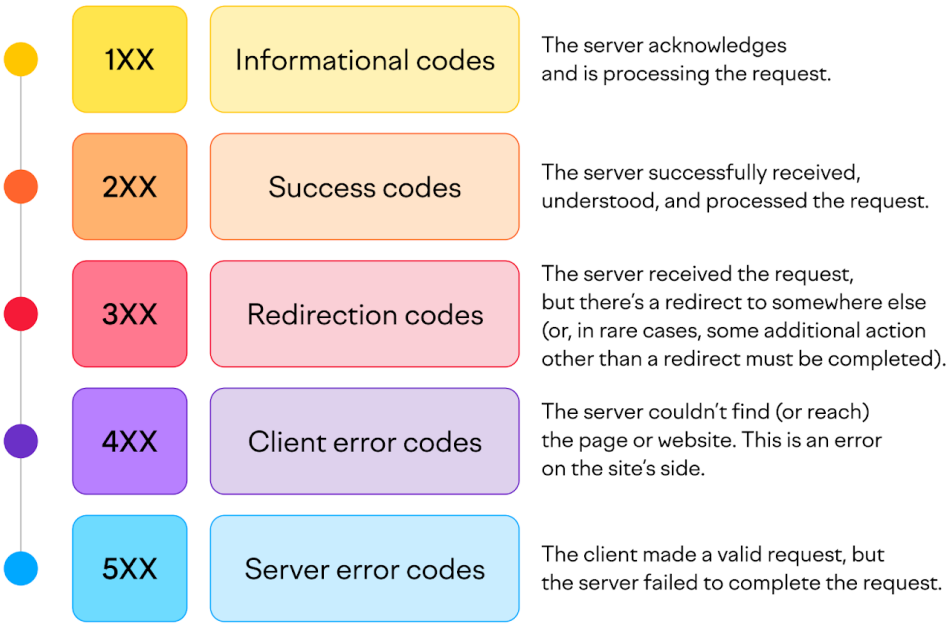
HTTP Response Status Codes

For more info → developer.mozilla.org/docs/Web/HTTP/Status

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

1. Informational responses (`100` – `199`)
2. Successful responses (`200` – `299`)
3. Redirection messages (`300` – `399`)
4. Client error responses (`400` – `499`)
5. Server error responses (`500` – `599`)

The status codes listed below are defined by RFC 9110.



Here's a fun way to remember these codes:

- **100–199:** Hold on, I'm thinking...
- **200–299:** Here you go, buddy!
- **300–399:** Go away, I don't want to see you anymore.
- **400–499:** You messed up, dude.
- **500–599:** I messed up, sorry.

Middlewares

Middleware in Express.js refers to functions that execute during the lifecycle of a request to the Express server. These functions have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle.
Middleware functions can perform a variety of tasks, such as:

- Executing any code: They can perform operations like logging, authentication, parsing data, etc.
- Making changes to the request and the response objects: They can modify the request object (req) and response object (res), adding new properties or modifying existing ones.
- Ending the request-response cycle: They can send a response to the client, ending the request-response cycle.
- Calling the next middleware function: If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

Body Parser : Body-parser is a middleware module for Express.js that simplifies the process of parsing incoming request bodies before the handlers process them. This is particularly useful for handling form submissions, JSON payloads, and URL-encoded data.

- ***`body-parser`*** extracts the entire body portion of an incoming request stream and exposes it on ***`req.body`.*** Different parsers are available for different content types.

And very commonly this is used to handle form data.
See code file to it's working and implementation.

Morgan : Morgan is a popular HTTP request logger middleware for Node.js and Express.js applications. It simplifies the process of logging requests to your application, providing detailed information about each request, which is helpful for debugging and monitoring purposes.
Key Features of Morgan Predefined formats:

- Morgan provides several predefined logging formats such as combined, common, dev, short, and tiny, which cover a wide range of logging needs.
- Custom formats: You can define your own custom logging formats to capture specific details about requests.
- Stream support: Morgan can write logs to various destinations, including files and streams.
- Immediate logging: It supports immediate logging of request data as soon as the request is received.

Custom Middleware : Creating custom middleware in Express.js is straightforward.
Middleware functions have access to the request object (req), the response object (res), and the next function, which is used to pass control to the next middleware in the stack.
Steps to Create Custom Middleware :

- Define the Middleware Function: Create a function that takes req, res, and next as arguments.
- Perform Operations: Inside the function, you can perform any operation like logging, modifying request/response objects, or handling authentication.
- Call next(): Call next() to pass control to the next middleware function. If you don't call next(), the request will be left hanging.