NodeJS

Express

Content

- Backend Basics
- RESTful APIs & HTTP
- Node.js
- Node Modules
- Express Framework
- MongoDB & Mongoose ORMs
- CRUD Operations
- Testing APIs
- Architectural patterns
- Routing
- Middlewares

Backend Basics

What Is Backend Development?

- Backend Development involves the logic, database, and other operations that are built behind the scenes to run the web servers efficiently.
- Backend Development refers to the server-side development of the web application.
- It is the part of the application where the server and database reside and the logics is built to perform operations.
- It includes the main features and functionalities of the application on the server.

Web Servers

- The web server runs websites, it stores, processes, and delivers (response) web pages to the user’s request.
- When the user makes a request by the web server, it is accepted by an HTTP server which finds and sends back the content to the browser through HTTP.
- Some examples of web servers are Apache and NGINX which are open-source platforms used to deliver content as per requests made.

Web Servers

- A static web server or stack consists of a computer (hardware) with an HTTP server (software). We call it “static” because the server sends its hosted files as it is to your browser”. It response/server static content.

- A dynamic web server consists of a static web server plus extra software, most commonly an application server and a database.

- We call it “dynamic” because the application server updates the hosted files before sending them to your browser via the HTTP server.

RESTful APIs & HTTP

RESTful APIs

- A RESTful API, or REST API, is an architectural style for designing networked applications.
- An API (Application Programming Interface) acts as a messenger, allowing different software applications to communicate and exchange data.
- It uses standard HTTP methods to access and manipulate resources, making it a popular choice for building web services and facilitating communication between different software systems.

HTTP Methods

- HTTP methods, also known as HTTP verbs, define the type of action to be performed on a resource. The most common methods are GET, POST, PUT, DELETE, and PATCH. These methods are fundamental to how clients interact with servers in web development and API design.

1. GET: Retrieves data from a specified resource.
2. POST: Sends data to a server to create or update a resource.
3. PUT: Replaces a resource with the provided data.
4. DELETE: Removes a resource.
5. PATCH: Partially modifies a resource.

HTTP Status Codes

- HTTP status codes are standardized responses from web servers to client requests, indicating the outcome of the request. They are categorized into five classes: informational (1xx), successful (2xx), redirection (3xx), client errors (4xx), and server errors (5xx).

1. 1xx - Informational: The server has received the request and is continuing to process it.

2. 2xx - Successful: The request was successfully received, understood, and accepted by the server. Examples include:

   - 200 OK: The request was successful.
   - 201 Created: The request was successful and a new resource was created.
   - 204 No Content: The request was successful, but there is no content to send in the response.

1.

HTTP Status Codes

3. 3xx - Redirection: The client needs to take further action to complete the request. Examples include:

   - 301 Moved Permanently: The requested resource has been moved to a new, permanent URI.
   - 302 Found: The requested resource has been temporarily moved to a different URI.

4. 4xx - Client Error: The request contains bad syntax or cannot be fulfilled due to an error on the client's part. Examples include:

   - 400 Bad Request: The server could not understand the request due to invalid syntax.
   - 401 Unauthorized: The request requires authentication.
   - 403 Forbidden: The server understood the request but refuses to authorize it.
   - 404 Not Found: The server cannot find the requested resource.
   - 429 Too Many Requests: The user has sent too many requests in a given amount of time.

1.

RESTful APIs

- Each request from a client to the server contains all the information needed to understand and process it, without the server needing to store any client-specific session information. (Stateless)

- REST APIs operate on a client-server model, where the client and server are independent and can evolve separately. (Client-Server Architecture)

- Responses from the server can be cached by clients or intermediary systems, improving performance and reducing server load. (Cacheable)

RESTful APIs

- How it works:

  1. Client Sends a Request:
  A client (e.g., a web browser, mobile app) sends an HTTP request to the server, specifying the desired resource and the action to be performed (e.g., GET to retrieve data, POST to create data).

  2. Server Processes the Request:
  The server receives the request, performs the requested action, and generates a response.

  3. Server Sends a Response:
  The server sends back an HTTP response, typically in JSON or XML format, containing the requested data or information about the result of the operation.

RESTful APIs

1. Why Use RESTful APIs:
    1. Simplicity:
        - REST APIs are relatively simple to understand and implement compared to other architectural styles.
    2. Flexibility:
        - RESTful APIs can be used with various programming languages and platforms, making them highly adaptable.
    3. Scalability:
        - The stateless nature of REST APIs makes them easy to scale to handle a large number of clients.

NodeJS

NodeJS

- Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to execute JavaScript code outside of a web browser. It is built on Google Chrome's V8 JavaScript engine, which is known for its high performance.

- It provides the necessary environment and tools to run JavaScript code on a server or as a standalone application, rather than solely within a browser.

- Node.js uses a non-blocking, event-driven I/O model, making it highly efficient for handling concurrent connections and real-time applications like chat applications or streaming services.

NodeJS

- It enables the use of JavaScript for backend development, allowing developers to use a single language for both client-side and server-side logic.
- Its architecture is designed for building scalable network applications that can handle a large number of simultaneous connections with reduced memory footprint.
- Node.js comes with NPM, a vast ecosystem of open-source libraries and tools that simplify development and extend its capabilities.
- Fun fact: more than 35% of professional developers use Node.js for building applications, including its libraries, frameworks, and tools.

NodeJS

- Common uses of Node.js:

  1. Building RESTful APIs and web servers.

  2. Creating real-time applications like chat apps and online games.

  3. Developing command-line tools and desktop applications.

  4. Building microservices and serverless functions.

NodeJS

- Starting a Node.js app:

1. Create a Project Directory: Create a new, empty directory where your Node.js project will reside.

```
$ mkdir my-node-app
$ cd my-node-app
```

2. Initialize with npm: Use the Node Package Manager (npm) to initialize your project and create a package.json file.

```
$ npm init
```

NodeJS

- Editing your package.json file:

```
{
"type": "module",     // means your project is using ECMAScript Modules (ESM) instead of CommonJS
"name": "cyberguardx_backend",   // your project name
"version": "1.0.0",                        // project versioning
"description": "",                         // project description
"main": "index.js",                        // where to start your app

"scripts": {
"test": "echo \"Error: no test specified\" && exit 1",
"serve": "node index.js",
},
```

NodeJS Modules

NodeJS Modules

- A NodeJS module is a separate file containing code that can be imported and reused in other parts of the application.

- It helps break down large applications into smaller, manageable sections, each focused on a specific functionality.

- By using modules, developers can keep code organized, reusable, and maintainable.

- Modules can contain:
  1. Variables
  2. Functions
  3. Classes
  4. Objects

NodeJS Modules

- NodeJS provides two primary module systems:

1. ES6 Modules (ECMAScript Modules - ESM):

    - Uses import to import modules.
    - Uses export to export functions, objects, or variables.
    - Modules are loaded asynchronously, allowing better performance.
    - Requires "type": "module" in package.json.

```
export function add(a, b) {
   return a + b;
}
export const PI = 3.1415;
```

$\longrightarrow$

```
import { add, PI } from './math.js';
console.log(add(2, 3));
```

NodeJS Modules

- Use Cases of ES6 Modules:

  1. Default Export & Import:

     - The default export allows a module to export a single function, object, or class as its main functionality. When importing, the name can be customized, making it more flexible than named exports.

```
export default function greet(name) {
  return ` Hello, ${name}! ` ;
}
```

```
import greet from './greet.js';
console.log(greet('Node.js'));
```

NodeJS Modules

- 2. Named Export With Aliases:
    - Named exports allow multiple functions, objects, or variables to be exported from a single module. Unlike default exports, named exports must be imported using the exact name they were exported with, unless an alias is provided during import.

```
export function multiply(a, b) {
   return a * b;
}
export function divide(a, b) {
   return a / b;
}
```

```
import { multiply as mul, divide as div }

from './operations.js';
console.log(mul(6, 3));
console.log(div(10, 2));
```

NodeJS Modules

2. CommonJS Modules (CJS)

- CommonJS is the default module system used in NodeJS. It enables code modularity by allowing developers to export and import functions, objects, or variables using module.exports and require().

```
//export
module.exports = { module1,
module2, ... };
```

```
//import

const module1 = require('./module1');
```

NodeJS Modules

2. CommonJS Modules (CJS)

- Uses require() to import modules.
- Uses module.exports to export functions, objects, or variables.
- Modules are loaded synchronously, meaning execution waits until the module is fully loaded.
- It is default in NodeJS, but not natively supported in browsers.
- Each module runs in its own scope, preventing variable conflicts.

Express.js

Express.js

- What is Express?:
  - Express.js, is a fast and minimalist web application framework for Node.js. It simplifies the process of building server-side applications and APIs by providing a robust set of features for handling HTTP requests, routing, middleware, and more.
  - It provides a core set of features for web development without imposing a rigid structure, allowing developers flexibility in how they build applications.
  - Express offers mechanisms to define routes based on URL paths and HTTP methods (GET, POST, PUT, DELETE, etc.), directing incoming requests to the appropriate handlers

Express.js

- What is Express?:
  - It supports the use of middleware functions, which are executed in a specific order during the request-response cycle. Middleware can perform tasks like logging, authentication, parsing request bodies, and more.
  - Express is widely used for building RESTful APIs, providing tools to manage requests and responses efficiently.
  - By simplifying server-side development and promoting modularity through middleware, Express helps create scalable and maintainable web applications.

Express.js

- Installing Express:
    - Express is a Node.js module, it can be installed using `$ npm install express` in your terminal.
- Importing Express:
    - Express can be imported like any other import through your index.js file, which will be your main start of the application

```
import express from 'express'    // importing
const app = express()            // initializing the app
```

Express.js

- Since we will be dealing with RESTful APIs, we need to make sure our express app can correctly parse and deal with json format:

```
app.use(express.json());                                  // Add this in your index.js file
```

- Let test if everything is correct by adding our first API and forwarding our app to port 3000:

```
app.get('/', (req, res) => {      // we are adding the client's request and the server's response
                                  as a parameter

res.send('Hello World!');});      // managing the response from the server

                                  // starting your app on port 3000
app.listen(3000, () => {
console.log('app listening on port 3000!');});
```

Express.js

- To start your server run $ npm run serve in your terminal to run your script
- Now head to http://localhost:3000/ on your browser to check your server's response
- To restart server automatically every time we make changes to our codebase, we can use the nodemon module for better productivity by using $ npm i nodemon –D
- Now update your package.json scripts value to include your dev environment with nodemon

```
"scripts": {
"test": "echo \"Error: no test specified\" && exit 1",
"serve": "node index.js",
"dev": "nodemon index.js"
},
```

MongoDB

MongoDB

- MongoDB is a popular, open-source, NoSQL database that uses a document-oriented approach to store data.
- It's known for its flexibility and scalability, making it well-suited for modern web and mobile applications.
- Instead of tables and rows like traditional relational databases, MongoDB stores data in flexible, JSON-like documents within collections.
- MongoDB is designed for high availability and horizontal scaling, making it suitable for handling large volumes of data and high traffic loads.

SQL VS NoSQL

- Relational Databases (SQL):
    - Data Model: Data is organized into tables with rows (records) and columns (fields).
    - Schema: Relational databases have a predefined schema, meaning the structure of the data (tables, columns, relationships) is fixed.
    - Relationships: Data is linked through defined relationships between tables, enabling complex queries and analysis.
    - Transactions: They excel at handling complex transactions with ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring data integrity.

SQL VS NoSQL

- Relational Databases (SQL) cont. :
  - Query Language: Relational databases use SQL (Structured Query Language) for data manipulation and retrieval.
  - Scalability: Typically scale vertically by upgrading server resources (CPU, RAM).
  - Examples: MySQL, PostgreSQL, Oracle, SQL Server.
  - Use Cases: Suitable for applications requiring strong consistency, complex transactions, and structured data, such as banking, e-commerce, and financial systems.

SQL VS NoSQL

- **Non-Relational Databases (NoSQL):**
  - **Can use various models like document (JSON), key-value, graph, or column-family, offering flexibility.**
  - **Schema: NoSQL databases generally have dynamic or flexible schemas, allowing for unstructured or semi-structured data.**
  - **Relationships: Relationships between data points are not as strictly defined as in relational databases.**
  - **Transactions: May prioritize scalability and performance over strict ACID compliance, with some offering eventual consistency.**

SQL VS NoSQL

- **Non-Relational Databases (NoSQL) cont. :**
  - **Query Language: NoSQL databases use various query languages or APIs, often specific to the database system.**
  - **Scalability: Often designed for horizontal scalability, distributing data across multiple servers for handling large volumes and high traffic.**
  - **Examples: MongoDB, Cassandra, Redis, Couchbase.**
  - **Use Cases: Ideal for applications dealing with large volumes of unstructured or semi-structured data, real-time analytics, social media, and content management systems.**

MongoDB

- Starting with MongoDB:
  - Installing: MongoDB is open-source and can be downloaded by following the installation documentation for your Linux distro at:

    https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-ubuntu/
  - After installing, lets access the cli by typing `$ mongosh` in your terminal
  - MongoDB runs on your localhost on port 27017 by default
  - Install MongoDB compass for an easier GUI interface

MongoDB

- Accessing your data:
  - Run **$ use <your_app_db_name>** in your mongosh shell
  - If the database <your_app_db> doesn't exist yet, MongoDB will create it as soon as you insert data.
  - Now create a collection and insert one document:

```
db.products.insertOne({                    // use .insertOne() to insert an object into your document
  name: "Test Product",
  price: 100,                              // then enter your product properties
  category: "Test Category"
})
```

MongoDB

mongoDB

- Using Mongoose in Express:
  - To install the mongoose module in your express app, simply run: $ npm i mongoose
  - Now back in our index.js file, connect to MongoDB using mongoose:

```
mongoose.connect('mongodb://localhost:27017/<your_app_db_name>').then(() => {
console.log('Connected to MongoDB');
app.listen(3000, () => {
console.log('app listening on port 3000!');
});
});
```

MongoDB

- Creating our first model:
    - Create a new file for our product model, and call it product.model.js (mongoose naming convetion)

```
import mongoose from 'mongoose'

const productSchema = new mongoose.Schema({
name: {
type: String,
required: [true, 'Product name is required'],
unique: true,
}});

export const Product = mongoose.model('Product', productSchema)
```

MongoDB

mongoDB

- Now that we have created our products model and exported it, we can now access it from anywhere in our code base by simply using:

```
import { Product } from '../models/product.model.js';
```

- We can also use mongoose helpers on our product model to use in our CRUD operations:

```
Model.deleteMany()
Model.deleteOne()
Model.find()
Model.findById()
Model.findByIdAndDelete()
Model.create()
Model.findByIdAndUpdate()
Model.findOne()
Model.findOneAndDelete()
Model.findOneAndReplace()
Model.findOneAndUpdate()
Model.replaceOne()
Model.updateMany()
Model.updateOne()
```

CRUD Operations

CRUD Operations

- CRUD is an acronym that stands for Create, Read, Update, and Delete. These four basic functions are the backbone of almost every web application
- Now that we are connected to our database and created our model, lets add basic CRUD APIs to manage our data:

```
app.post('/api/products', async (req, res) => {      // async POST API
try {
const newProduct = await Product.create(req.body);  // await for creation
res.status(201).send({ message: 'Product created', product: newProduct });  // return response
} catch (err) {
res.status(500).send({ message: err.message });  // error handling
}
});
```

CRUD Operations

- Now to view our entries, Let's create a GET API

```
   app.get('/api/products', async(req, res) => {           // asynchronous promise
try {
const allProducts = await Product.find();                  // .find() returns all products
res.status(200).json(allProducts);                         // return response as JSON
} catch (err) {                                            // handling errors
res.status(500).send({ message: err.message });
}
});
```

CRUD Operations

- After making sure our data was persisted, we can create a PUT API to update it:

```
 app.put('/api/products/:id', async(req, res) => {
try {
const updatedProduct = await Product.findByIdAndUpdate(req.params.id, req.body, { new: true });
if (!updatedProduct) {
return res.status(404).send({ message: 'Product not found' });
}
res.status(200).send({ message: 'Product updated', product: updatedProduct });
} catch (err) {
res.status(500).send({ message: err.message });
}});
```

CRUD Operations

- Finally, let's try deleting our entry by creating a DELETE API:

```
app.delete('/api/products/:id', async(req, res) => {
try {
const deletedProduct = await Product.findByIdAndDelete(req.params.id);
if (!deletedProduct) {
return res.status(404).send({ message: 'Product not found' });
}
res.status(200).send({ message: 'Product deleted' });
} catch (err) {
res.status(500).send({ message: err.message });
}
});
```

Testing APIs

APIs Testing

POSTMAN

- What is API Testing? :
  - API testing is a process that confirms an API is working as expected. There are several types of API tests, and each one plays a distinct role in ensuring that the API's functionality, security, and performance remain reliable.
  - Developers can run API tests manually, or they can automate them with an API testing tool.
  - Traditionally, API testing has occurred at the end of the development phase, but an increasing number of teams are running tests earlier in the API lifecycle.

APIs Testing

POSTMAN

- What is API Testing? :

    - API testing is a process that confirms an API is working as expected. There are several types of API tests, and each one plays a distinct role in ensuring that the API's functionality, security, and performance remain reliable.

    - Developers can run API tests manually, or they can automate them with an API testing tool.

    - Traditionally, API testing has occurred at the end of the development phase, but an increasing number of teams are running tests earlier in the API lifecycle.

APIs Testing

- What are the different types of API testing? :

1. Functional testing

    - API functional testing verifies that an API meets its specified requirements. This type of testing involves sending specific requests to the API, analyzing the responses, and comparing the actual outcomes with the expected results to ensure the API performs as designed.

2. Integration Testing

    - API integration testing is a critical step in ensuring that different parts of a system are compatible with one another. It helps confirm that APIs can reliably and efficiently communicate and transfer data between one another—even as they evolve over time.

APIs Testing

POSTMAN

3. Load Testing:
    - API load testing enables developers to confirm whether their API is able to operate reliably during times of peak traffic. It typically involves using a testing tool to simulate large request volumes and measure the resulting response times and error rates. This type of testing is often performed in anticipation of a significant load increase, such as right before a product launch or yearly sale.
4. Unit Testing:
    - API unit testing is the process of confirming that a single endpoint returns the correct response to a given request. Unit tests may validate that an endpoint handles optional parameters correctly, or that it returns the appropriate error message when sent an invalid request.

APIs Testing

3. Load Testing:
    - API load testing enables developers to confirm whether their API is able to operate reliably during times of peak traffic. It typically involves using a testing tool to simulate large request volumes and measure the resulting response times and error rates. This type of testing is often performed in anticipation of a significant load increase, such as right before a product launch or yearly sale.
4. Unit Testing:
    - API unit testing is the process of confirming that a single endpoint returns the correct response to a given request. Unit tests may validate that an endpoint handles optional parameters correctly, or that it returns the appropriate error message when sent an invalid request.

APIs Testing

POSTMAN

- What is Postman :
  - Postman: Postman is an API development tool that helps to build, test and modify APIs.
  - It has the ability to make various types of HTTP requests(GET, POST, PUT, PATCH).
  - Collections in Postman serve as a powerful organizational tool, allowing developers to categorize and manage API requests efficiently.
  - This organized structure facilitates seamless sharing and collaboration within development teams.

APIs Testing

POSTMAN

- Starting with Postman :
    - To use Postman to test an API, start by creating a new request, naming and saving it within a collection. Set up the request by specifying the API URL, choosing the HTTP method, and adding parameters, headers, authentication, and body as needed.
    - Organize your requests by creating folders within collections.
    - Run the request, view the results, and optionally, write tests for automation.
    - Save and share your work, and export collections if needed.

Architectural patterns

Architectural Patterns

- What is an architectural pattern? :

  - Architectural patterns are standard ways of organizing a software system’s structure, how components like databases, APIs, UI, services, etc. interact and are layered.

  - They're like blueprints for building software that:

    - Solves common problems
    - Makes code more scalable, maintainable, and understandable

Architectural Patterns

- Types of architectural patterns :

  - There are a lot of architectural patterns that a developer can use, for example:

    - MVC (Model View Controller)
    - MSC (Model Service Controller)
    - Microservices Architecture

MVC Architecture

- MVC, or Model-View-Controller, is a software architectural pattern that divides an application into three interconnected parts: the Model, the View, and the Controller.

- This separation of concerns helps create more organized, maintainable, and testable applications, particularly user interfaces.

- Model:
  - Represents the application's data and business logic. It manages data, states, and rules, and handles interactions with the database. For example, in a booking system, the model might handle user data, booking details, and price calculations.

MVC Architecture

- View:
  - Responsible for presenting the data to the user. It takes data from the model and displays it in a user-friendly format. For instance, a view could display a list of available rooms or a user's profile information.

- Controller:
  - Acts as an intermediary between the Model and the View. It handles user input, updates the model, and selects the appropriate view to display. For example, if a user submits a form, the controller would receive the input, update the model with the new data, and then render a view to confirm the action.

MVC Architecture

- Benefits of using MVC:
  - Improved organization and maintainability:
    - Separating concerns into distinct components makes it easier to understand, modify, and debug code.
  - Increased testability:
    - Each component can be tested independently, allowing for more thorough testing and fewer integration issues.
  - Enhanced collaboration:
    - Developers can work on different components simultaneously, leading to faster development cycles.

MVC Architecture

- Benefits of using MVC cont. :
    - Better scalability:
        - MVC's modular structure allows for easier scaling of the application as it grows.
    - Code reuse:
        - Components can be reused in different parts of the application or even in other projects.
    -

MSC Architecture

- MSC is a software architectural pattern commonly used in backend API development. It organizes an application into three distinct components: the Model, the Service, and the Controller.

- This separation of concerns helps improve code readability, maintainability, and scalability — particularly in RESTful APIs or server-side logic where there's no visual UI.

- Model :
  - The Model represents the application's data structure and handles all interactions with the database.
  - It defines the schema and validation rules.
  - It is responsible for creating, retrieving, updating, and deleting data (CRUD).

MSC Architecture

- Service:

  - The Service layer contains the business logic of the application.
  - It processes data, applies rules, and performs operations based on business needs.
  - It acts as the brain of the application, sitting between the controller and model.
  - Keeps controllers thin and focused on input/output.

- Example:
  A service might validate product stock, apply a discount rule, or calculate shipping fees before interacting with the model.

MSC Architecture

- Controller:

  - The Controller manages incoming HTTP requests and responses.
  - It receives requests (like POST, GET, PUT, DELETE),
  - Passes the data to the appropriate service function,
  - Then returns a response (JSON) to the client.

- Example:
  When a user sends a POST request to create a product, the controller collects the request body, calls the service to create the product, and returns a JSON response.

MSC Architecture

- Benefits of Using MSC :
  - Improved organization and separation of concerns:
    - Each part of your app has a clear responsibility:
    - Models handle data
    - Services handle logic
    - Controllers handle HTTP communication
  - Increased testability :
    - Test your business rules independently of routes
    - Mock data easily without depending on database operations or HTTP

MSC Architecture

- Benefits of Using MSC cont. :
  - Better scalability and flexibility
    - As the app grows, adding new features becomes easier:
    - You can reuse services across multiple controllers
    - You can refactor logic without breaking routes
  - Reusability of logic
    - The service layer allows for reusing business logic in:
      - Multiple endpoints
      - Scheduled jobs
      - CLI tools or background workers

Microservices

- What is Microservices? :

  - Microservices is a software architectural pattern where an application is built as a collection of small, independent services — each responsible for a specific piece of functionality.

  - Each service is self-contained, can be developed and deployed independently, and communicates with other services through APIs or messaging systems.

  - Solves one business problem

  - Has its own database or data store

Microservices

- Benefits of Microservices:
  - Independent development & deployment:
    - Teams can build, test, and deploy services separately
    - No need to redeploy the whole app for a single update
  - Scalable by service:
    - You can scale only the parts that need it
    - e.g. scale the “payment service” separately from “admin service”

Microservices

- Benefits of Microservices:
  - Technology flexibility:
    - Each service can use a different language or database
    - e.g. Node.js for users, Python for ML service, Go for performance-critical logic
  - Better fault isolation:
    - If one service fails (e.g., search), others (e.g., checkout) still work
    - Easier to debug issues within isolated services

Routing

Routing

- What Is the Routers Layer?
  - The routers layer in Express.js helps you organize and group your API routes by feature or domain (e.g., /users, /products, /orders).
  - It acts as a middle layer that:
    - Matches HTTP routes (URLs + methods)
    - Delegates the request to the correct controller function
    - Keeps your index.js file clean and scalable

Routing

- We start by adding a routes folder, which will contain all of our project's routes.

```
import * as productController from '../controllers/product.controller.js';
Import express from 'express'

const router = express.Router();
```

- We will then start using this router in our index file to make it cleaner and more maintainable.

```
import productRouter from './routes/product.route.js';     // import your Router
app.use('/api/products', productRouter);                    // use it for all requests that starts with the given URL
```

Routing

- Back in our router, we match the appropriate controller for each request:

- router.get('/', productController.getAllProducts);
- router.get('/:id', productController.getProductById);
- router.post('/', productController.createProduct);
- router.put('/:id', productController.updateProduct);
- router.delete('/:id', productController.deleteProduct);
- export default router;

Middlewares

Middlewares

- What Is Middleware in Express.js? :
  - Middleware is a function that runs between the request coming from the client and the response sent by the server.
  - You can think of it as a pipeline or layered chain where each middleware:
    - Can read or modify the request (req)
    - Can read or modify the response (res)
    - Can end the request/response cycle OR pass control to the next middleware

Middlewares

- Common Middlewares Use Cases:
- Logging: Logging incoming requests and responses for debugging and monitoring.
- Authentication and Authorization: Checking if a user is authenticated and authorized to access a resource.
- Data Parsing: Parsing request bodies (e.g., JSON, URL-encoded).
- Error Handling: Catching and handling errors that occur during request processing.
- Modifying Request/Response Headers: Adding or modifying headers to the request or response.

Middlewares

Middleware execution flow:

Incoming Request

↓

[Logging Middleware]

↓

[Authentication Middleware]

↓

[Route Handler (Controller)]

↓

[Error Handler (if needed)]

↓

Response Sent

Any Questions ?