MERN2 - Intro ap API, Express and CRUD

What is express
How to use express with node
Http methods-get,post,put,delete, patch
Postman-to test your api endpoints
Middlewares

What do we understand by API

1. Application programming interface
2. An API (Application Programming Interface) is a set of rules and definitions that allows software applications to communicate with each other. In the context of web development, APIs are typically exposed as endpoints on a server that clients (such as web browsers or mobile apps) can interact with to perform various operations.
3. In backend development, an API consists of various endpoints or routes that define specific functionalities or services that your server provides.
4. Each endpoint corresponds to a specific URL path and usually performs a particular action when accessed.
5. To get started with this , we learn about Express

Express Module

Express.js is a minimalist web framework for Node.js. While you can certainly build web servers and applications using just the built-in HTTP module in Node.js, Express.js provides a higher-level way to to do certain tasks like manage routes, requests, and views

This is same like how we can do all our front end development using vanilla js but with React, angular and other lib and framework, we end up writing less code and getting more functionality.

some benefits and reasons why developers often choose Express.js over vanilla Node.js for web development:

1. Simpler and cleaner syntax
2. Code readability
3. Routing: Express provides a straightforward way to define routes and handle different HTTP methods (GET, POST, PUT, DELETE, etc.). In vanilla Node.js, you'd typically have to handle URLs and methods manually, which can be cumbersome.
4. Middleware: One of the core features of Express is its convenience in use of middleware, which are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. .

5. Request handling / routing much more scalable and manageable
6. Lot of heavy lifting of logic and features are done by express

Setting Up an Express Application

1. First, you need to install Express in your project:
2. Initialise a new node project
    a. Create a file index.js
    b. Npm init -y
3. npm install express --save
4. Now, let's create a simple Express application:

```
// Import the Express module
const express = require('express');

// Create an Express application
const app = express();

// Define a route
app.get('/', (req, res) => {
 res.send('Hello, Express!');
});

// Start the server
const port = 3000;
app.listen(port, () => {
 console.log(`Server is running on port ${port}`);
});
```

We import the express module and create an Express application instance.

We define a route for the root URL ("/") using app.get(), which responds with "Hello, Express!" when accessed via a GET request.

We start the server on port 3000.

Example 2: Express Routes

Express allows you to define multiple routes for different URL paths and HTTP methods. Here's an example:

```
// Import the Express module
const express = require("express");

// Create an Express application
const app = express();

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

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

app.post("/data", (req, res) => {
 res.send("Received a POST request.");
});

// Start the server
const port = 3000;
app.listen(port, () => {
 console.log(`Server is running on port ${port}`);
```

```
});
```

To test the POST request, let us go use our POSTMAN

1. Creating a Request:
    a. Open Postman and click the "New" button to create a new request.
    b. Choose the HTTP method (e.g., GET, POST) for your request.
2. Request URL:
    a. Enter the URL of the API endpoint you want to test in the request URL field.
3. Headers:
    a. You can set request headers by clicking on the "Headers" tab.
    b. Headers are used to pass additional information to the server, such as authentication tokens or content type.
4. Request Body:
    a. If your request requires a request body (e.g., for POST or PUT requests), you can define it in the "Body" tab.
    b. You can send data in various formats like JSON, form-data, x-www-form-urlencoded, etc.
5. Sending a Request:
    a. Click the "Send" button to send the request to the specified endpoint.

    b. Postman will display the response in the lower part of the window.
6. Response:
    a. You can view the response status code, headers, and body in the response section.
    b. You can also format and highlight the response body using the options provided.
7. Collections:
    a. Organize your requests into collections for better management.
    b. Create a new collection by clicking the "New" button under "Collections."

Seeing the request body

```
app.post("/data", (req, res) => {
   console.log(req.body);
 res.send("Received a POST request.");
});
```

It prints undefined

To see the request body, we wiss use our first middleware

When building web applications with Express, you'll often need to handle data sent from clients (such as web browsers or mobile apps)

to your server. This data is usually sent in the form of JSON (JavaScript Object Notation).

So we use something like **app.use(express.json())**

What Does app.use(express.json()) Do?

Purpose: It helps your Express application understand and work with JSON data sent in requests.

Functionality: It automatically parses incoming JSON requests and makes the data available in req.body.

What is express.json()?

Purpose: The express.json() function is a built-in middleware in Express. Its main job is to parse incoming JSON data from the client and make it available in req.body.

Why Middleware?: Middleware functions run between receiving the request and sending the response. They can perform tasks like parsing data, handling authentication, logging, and more.
app.use is a method in Express that is used to apply middleware functions to our application

When we use app.use, you are essentially telling Express to use a particular middleware function. This function will be called for every incoming request, unless you specify a path to limit its scope.

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

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

app.post("/data", (req, res) => {
   console.log(req.body);
 res.send("Received a POST request.");
});
```

title: Https Methods

POST Request Example:

In this example, we'll create a basic Express application that handles a POST request to add a new user to a list of users.

```
const users = [
 { id: 1, name: "User 1" },
 { id: 2, name: "User 2" },
];

// POST endpoint to add a new user
app.post("/users", (req, res) => {
```

```
  const newUser = req.body;

  // Assign a unique ID to the new user (in a real app, you'd typically use a database)
  const userId = users.length + 1;
  newUser.id = userId;

  // Add the new user to the list
  users.push(newUser);

  res.status(201).json({ message: "User created", user: newUser });
});
```

To test this, send a POST request to http://localhost:3000/users with a JSON body, for example:

```
{
  "name": "New User"
}
```

DELETE Request Example:

Let us discuss about Route Parameters using which we will implement delete

Route parameters are part of the URL's path and are used to define variable parts of a route. They are typically denoted by placeholders in the route pattern, surrounded by colons (:). When a client makes a request with a URL that matches the route pattern, the values specified in the URL are extracted and made available to the server or application.

For example, in a RESTful API, you might have a route for retrieving a specific user's profile:

GET /users/:userId

In this URL, :userId is a route parameter. When a request is made to /users/123, the server can extract the value 123 from the URL and use it to retrieve the user with that ID.

```
// DELETE endpoint to delete a user by ID
app.delete("/users/:id", (req, res) => {
 const userId = parseInt(req.params.id);

 // Find the user index by ID
 const userIndex = users.findIndex((user) => user.id === userId);

 if (userIndex === -1) {
   return res.status(404).json({ message: "User not found" });
 }

 // Remove the user from the array
 users.splice(userIndex, 1);

 res.json({ message: "User deleted" });
});
```

To test this, send a DELETE request to http://localhost:3000/users/1 to delete the user with ID 1.

These examples demonstrate how to implement POST and DELETE requests in an Express.js application. They focus on the core functionality without extensive error handling or database interactions. In practice, you would typically include more robust validation and potentially use a database for data storage.

title: Middlewares

Middleware functions in Express.js are functions that have access to the request (req) and response (res) objects and can perform actions or transformations on them.

They are used to handle tasks like parsing request data, authentication, logging, error handling, and more.

Middleware functions can be added to your Express application using app.use() or applied to specific routes using app.use() or app.METHOD() (e.g., app.get(), app.post()).

1.
2. In the world of web development, especially with frameworks like Express.js in Node, "middleware" are just like checkpoints. When a request comes into your application, it doesn't just jump straight to the final destination. It often goes through several "checkpoints" or "middlewares" that can check the request, transform it, log it, validate it, and more.

3. So, in simple terms, middleware functions are the individual "checkpoints" in your application that requests go through before reaching their final destination.

Let us create a custom middleware. Put the below code, right at the top

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

const loggerMiddleware = (req, res, next) => {
   console.log(`[${new Date().toISOString()}] ${req.method}
${req.url}`);
   next(); // Call the next middleware in the chain
 };
  // Register the middleware globally for all routes
 app.use(loggerMiddleware);
```

When you call next() inside this middleware, you're telling Express to proceed to the next handler in the line for the current request.This "next handler" could be another middleware or a route handler

To ensure your logger middleware runs for every request, you should place app.use(loggerMiddleware) before any route handlers.

Calling middleware for specific routes

if you wanted to apply loggerMiddleware only to a specific route, you could do the following:

```
app.get('/special', loggerMiddleware, (req, res) => {
   res.send('This route is special!');
 });
```

Built-in Middleware

express.static(root, [options]):

express.static is a middleware function that serves static files such as HTML, CSS, JavaScript, images, and more.

It is often used to serve client-side assets, making it easy to host static files like HTML, CSS, and JavaScript for your web application.

The root parameter specifies the root directory from which to serve static files.

The optional options object allows you to configure various settings, such as caching and file handling.

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

In this example, if you have an "index.html" file in the "public" directory, you can access it in your browser by navigating to http://localhost:3000/index.html.

express.json([options]):

express.json is a middleware function that parses incoming JSON requests and makes the parsed data available in the req.body property.

express.urlencoded([options]):

express.urlencoded is a middleware function that parses incoming URL-encoded data from forms and makes it available in the req.body property.

Query Parameters

In web development, route parameters and query parameters are mechanisms for passing data to a web server or an application, typically through a URL.

They are commonly used to customize and control the behavior of a web application by providing information about the requested resource or specifying additional options.

Query parameters are part of the URL's query string and are used to provide additional information or data to the server. They are typically

specified after the ? character in the URL and are in the form of key-value pairs.

For example, in a search feature, you might have a URL that includes query parameters to filter results:

GET /search?q=keyword&page=2&sort=desc

In this URL, q, page, and sort are query parameters. They allow the server to understand the search query, the desired page, and the sorting order.

In summary, route parameters and query parameters are essential for building dynamic web applications and APIs. They allow you to customize the behavior of your routes and pass data between clients and servers effectively.

Route parameters are part of the URL's path and are extracted using placeholders in the route pattern, while query parameters are part of the URL's query string and are provided as key-value pairs after the ? character. Express.js simplifies the handling of both types of parameters in your server-side code.

Go to flipkart.com and search for a product.

Copy the url from the browser and paste in POSTMAN

See the Params in the POSTMAN

Let us implement a route to see query params

```
app.get("/search", (req, res) => {
 // Access query parameters using req.query
 const queryParams = req.query;
 console.log("Query Parameters:", queryParams);

 // Respond to the client
 res.send(`YOur parameters are: ${JSON.stringify(queryParams)}`);
});
```

Difference between put and patch

PUT: This method is used when you want to update a complete resource. When you make a PUT request, you provide a complete updated object. The server then replaces the existing resource with the provided object. If certain attributes are missing in the request, those are typically set to their default values or removed. Essentially, a PUT request entirely replaces the existing resource with a new version.

Use PUT when you have the complete updated state of the resource. PUT is idempotent, meaning that making the same request multiple times will result in the same state of the resource on the server.
PATCH: On the other hand, PATCH is used for partial updates. With a PATCH request, you only need to provide the specific changes to the

resource, not the complete resource. The server then applies these changes to the existing resource.

Use PATCH for updating parts of the resource or for situations where sending the complete resource is not feasible or necessary.
PATCH can be idempotent, but it's not a requirement. It depends on how the server processes the PATCH request.

404 pages

Suppose we need to implement a way that if no route matches or user enters a random path then a page not found message from the server should be sent

Break down the problem

1. We need a way to watch for every request
2. We need to ensure that it does not return a 404 for a valid route but if no handlers are found then this should run

Have this at the end

```
app.use((req, res) => {
 res.status(404).send("Page not found");
});
```

What happens if we move this to the top

```
const app = express();
app.use(express.json());
app.use(express.static("public"));

const loggerMiddleware = (req, res, next) => {
 console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
 next(); // Call the next middleware in the chain
};

// Register the middleware globally for all routes
//   app.use(loggerMiddleware);

app.use((req, res) => {
   res.status(404).send("Page not found");
 });

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

app.post("/data", (req, res) => {
 console.log(req.body);
 res.send("Received a POST request.");
});

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

Now for every request page not found will be thrown. Hence order of middleware and handlers matters