5. Introduction to Express.js

- What is Express.js and why use it?
- Setting up an Express.js application
- Basic routing in Express.js
- Creating RESTful routes

Introduction to Express.js

What is Express.js and Why Use It?

Express.js is a fast, minimal, and flexible web application framework for Node.js. It simplifies building web applications and APIs by providing robust features for handling HTTP requests, middleware, routing, and more.

Why Use Express.js?

1. **Simplicity**: Provides a straightforward API that simplifies the process of building server-side applications and APIs.
2. **Middleware Support**: Easily integrate middleware to handle requests, responses, and error handling.
3. **Routing**: Manages different routes (URLs) within your application, allowing complex URL structures.
4. **Community and Ecosystem**: Has a large community and extensive set of plugins and libraries.
5. **Scalability**: Ideal for building scalable applications with RESTful services and modern web applications.

Setting Up an Express.js Application

Install Node.js and NPM: Ensure Node.js and NPM are installed. You can check by running:

```
node -v

npm -v
```

Initialize a New Node.js Project: Create a project directory, initialize it with a package.json file, and install Express.

```
mkdir my-express-app

cd my-express-app

npm init -y
```

Install Express.js: Install Express as a dependency in your project.

```
npm install express
```

Write the Basic Express Application: In app.js, add the following code to create a simple server:

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

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

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

Basic Routing in Express.js

Routing in Express.js is essential for handling HTTP requests and defining responses for different paths in your application. Express provides an easy and flexible way to set up routes for various HTTP methods like GET, POST, PUT, and DELETE.

Defining Routes in Express.js

A route in Express.js is typically defined by specifying:

1. **Path**: The URL or endpoint the route will handle.
2. **HTTP Method**: The type of HTTP request (GET, POST, etc.).
3. **Callback Function(s)**: The function(s) executed when the route is matched.

```
app.METHOD(PATH, HANDLER);
```

METHOD: The HTTP method (e.g., get, post).
PATH: The endpoint URL (e.g., /users).
HANDLER: A function that runs when the route is accessed.

Examples of Basic Routes

1. **Setting up a Basic Express App**

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

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

GET Route Handles GET requests for a specific path.

```
app.get('/', (req, res) => {
  res.send('Welcome to the Home Page');
});
```

POST Route Handles POST requests, typically used for submitting data.

```
app.post('/submit', (req, res) => {
  res.send('Form submitted successfully');
});
```

PUT Route Handles PUT requests, often used for updating existing data.

```
app.put('/update', (req, res) => {
  res.send('Data updated');
});
```

DELETE Route Handles DELETE requests, commonly used for deleting resources.

```
app.delete('/delete', (req, res) => {
  res.send('Data deleted');
});
```

Using Route Chaining for the Same Path

Express allows chaining multiple HTTP methods for the same path.

```
app.route('/items')
  .get((req, res) => {
    res.send('Get a list of items');
  })
  .post((req, res) => {
    res.send('Add a new item');
  })
  .put((req, res) => {
    res.send('Update an existing item');
  })
  .delete((req, res) => {
    res.send('Delete an item');
  });
```

Creating RESTful Routes in Express.js

RESTful routes follow a standard pattern to handle CRUD (Create, Read, Update, Delete) operations on resources, making web APIs more intuitive and easier to work with. In Express.js, creating RESTful routes allows you to map specific HTTP methods (GET, POST, PUT, DELETE) to standard URL structures.

RESTful Route Structure

A typical RESTful API for a resource (e.g., "users") includes the following routes:

HTTP Method	URL	Description
GET	/users- get-data	Retrieve a list of users
POST	/users -post-data	Create a new user
PUT	/users -update-data /:id	Update an existing user
DELETE	/users -delete-data /:id	Delete an existing user

Setting Up RESTful Routes in Express.js

Define RESTful Routes for a Resource (e.g., "users")

Here’s how you can implement the RESTful routes for a “users” resource.

```
// console.log("Express")

let express  = require('express')
let app = express()
app.use(express.json());
let users = [
    {id:1,name:'Hema'},
    {id:2, name:'Mahesh'},
    {id:3, name:'TATA'}

]
app.get("/users-get-data", (req, res) => {
  // console.log(req,res)
  console.log(req);
  // res.json(users)
  res.send(
    // `<h1>${users[0].name}</h1>
    // <h1>${users[1].name}</h1>
    // <h1>${users[2].name}</h1>
    // `
    `<div>${users.map((data) => {
      return `<h2>ID : ${data.id}</h2>
                <h1>Name : ${data.name}</h1>`;
    })}</div> `
  );
});

app.post('/users-post-data',(req,res)=>{
    console.log(users)
    // res.json(users)
    newUsersData = {
        id:req.body.id,
        name:req.body.name
    }
    console.log(newUsersData)
    users.push(newUsersData)
    res.json(users)
})

app.put('/users-update-data/:id',(req,res)=>{
        let userId = parseInt(req.params.id)
        console.log(req.body)
        console.log(userId)

        let {name} = req.body;
        let userIndex = users.findIndex((user)=>{
            console.log(user.id === userId)
            return user.id === userId
        })
        console.log(userIndex)
        users[userIndex].name = name
        console.log(`User with ID ${userId} updated:`, users[userIndex]);
        res.json(users)
})

app.delete('/users-delete-data/:id',(req,res)=>{
    let userId = parseInt(req.params.id)
    console.log(req.body)
    console.log(userId)
    let userIndex = users.findIndex((user)=>{
        console.log(user.id === userId)
        return user.id === userId
    })
    console.log(userIndex)

       let deletedUser =  users.splice(userIndex,1)
       console.log(`User with ID ${userId} deleted:`, deletedUser[0]);
      res.json(users)
})
let PORT = 3000;
app.listen(PORT,()=>{
    console.log(`server is starting on http://localhost:${PORT}`)
})
```

GET /users: Returns all users as a JSON array.
GET /users/:id: Retrieves a single user by ID. If the user doesn't exist, it returns a 404 Not Found status.
POST /users: Adds a new user to the users array. It expects a JSON body (e.g., { "name": "Charlie" }).

PUT /users/:id: Updates an existing user's name based on the ID. If the user doesn't exist, it returns 404 Not Found.
DELETE /users/:id: Deletes a user by ID. If the user doesn't exist, it returns 404 Not Found.

Interview Questions:

What is Express.js and Why Use It?

Answer:
Express.js is a fast, lightweight, and flexible Node.js framework used for building web applications and APIs. It provides a robust set of features to handle HTTP requests, manage middleware, route handling, and more.

Benefits of Using Express.js:

- **Simplicity and Flexibility**: Makes it easy to handle complex routing, middleware, and request handling with a simple API.
- **Middleware Support**: Allows adding middleware to process requests and responses, handling functionalities like authentication, logging, and data parsing.
- **Routing**: Simplifies route management with built-in functions to handle various HTTP methods.
- **Large Ecosystem**: Has a large community and a rich ecosystem of plugins and third-party libraries.

How Do You Set Up an Express.js Application?

Answer:
To set up an Express application, follow these steps:

1. **Initialize a Node.js Project**:

```
mkdir my-express-app

cd my-express-app

npm init -y
```

2. Install Express:

```
npm install express
```

3. Create the Application File (e.g., app.js): In app.js, import Express, create an Express instance, and define a basic route:

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

app.get('/', (req, res) => res.send('Hello, Express!'));
app.listen(3000, () => console.log('Server running on http://localhost:3000'));
```

4. Run the Application:

node app.js

Explanation:

- **express()**: Creates an Express application instance.
- **app.get()**: Sets up a route that listens for GET requests at the root path /.
- **app.listen()**: Starts the server on port 3000.

What is Basic Routing in Express.js?

Answer:
Basic routing in Express.js refers to defining routes that respond to HTTP requests at specific endpoints. Each route can handle a specific HTTP method (GET, POST, etc.) and perform certain actions.

Example

```
app.get('/about', (req, res) => res.send('About Page'));
app.post('/submit', (req, res) => res.send('Form Submitted'));
```

Explanation:

- **app.get()**: Handles GET requests. The above example sends "About Page" in response to /about.
- **app.post()**: Handles POST requests, often used to submit data, such as form data.

Usage: Basic routing is suitable for simple applications with single or few routes, such as static websites or small prototypes.

How Do You Create RESTful Routes in Express.js?

Answer:
RESTful routes follow REST principles, mapping specific HTTP methods to CRUD operations on resources. This makes the API predictable and standardized.

HTTP Method	URL	Description
GET	/users	Get a list of users
GET	/users/:id	Get a specific user
POST	/users	Create a new user
PUT	/users/:id	Update an existing user

HTTP Method	URL	Description
DELETE	/users/:id	Delete a user

What’s the Difference Between Basic Routing and RESTful Routing?

Answer:

- **Basic Routing**: Focuses on responding to HTTP requests for individual endpoints without enforcing a structured approach. Ideal for simple applications.
- **RESTful Routing**: Adheres to REST principles, mapping routes to CRUD operations on resources. RESTful routing is used for building structured and predictable APIs.

How Does the app.route() Method Simplify Routing?

Answer:
The app.route() method allows chaining route handlers for the same path, reducing redundancy in code by grouping multiple HTTP methods for a specific route.

```
app.route('/items')
  .get((req, res) => res.send('Get items'))
  .post((req, res) => res.send('Add item'))
  .put((req, res) => res.send('Update item'))
  .delete((req, res) => res.send('Delete item'));
```

In this example, app.route('/items') groups all HTTP methods for the /items path, making the code cleaner and more manageable.