Part - 4

Node.js

Modules

Module – 1 Introduction to Node.js

- 1.1 What is Node.js?
- 1.2 Features and Advantages of Node.js.
- 1.3 Installation and Setup of Node.js.
- 1.4 Your first Node.js "Hello World" Program.

Module – 2 Node.js Modules and NPM

- 2.1 Understanding Node.js Modules.
- 2.2 Creating and using Custom modules.
- 2.3 Introduction to the Node Package Manager (NPM).
- 2.4 Installing and Managing Packages with NPM.

Module – 3 Asynchronous JavaScript

- 3.1 Asynchronous Programming in Node.js.
- 3.2 Callbacks and the Event Loop.
- 3.3 Promises and Async/Await for better Asynchronous code.

Module – 4 File System and Streams

- 4.1 Working with the File System in Node.js.
- 4.2 Reading and Writing Files.
- 4.3 Using streams for Efficient Data Processing.

Module – 5 HTTP and Web Servers

- 5.1 Creating HTTP Servers with Node.js.
- 5.2 Handling HTTP Requests and Responses.
- 5.3 Building RESTful APIs with Node.js.

Module – 6 Express.js Framework

- 6.1 Introduction to Express.js.
- 6.2 Creating Web Applications using Express.js.
- 6.3 Routing, Middleware, and Template Engines.

Module – 7 Databases and NoSQL with Node.js

- 7.1 Integrating Databases (SQL and NoSQL) with Node.js.
- 7.2 Using MongoDB as a NoSQL Database.
- 7.3 Performing CRUD Operations with Databases.

Module – 8 Authentication and Authorization

- 8.1 Implementing User Authentication in Node.js.
- 8.2 Role-Based Access Control.
- 8.3 Securing Node.js Applications.

Module – 9 RESTful API Development

- 9.1 Designing and Developing RESTful APIs in Node.js.
- 9.2 Handling Requests, Validation, and Error Responses.
- 9.3 API Documentation and Testing.

Module – 10 Real-time Applications with WebSocket

- 10.1 Understanding WebSocket and Real-Time Communication.
- 10.2 Develop Real-Time Applications with Node.js - WebSocket.
- 10.3 Building a Chat Application as a Practical Example.

Module – 11 Scaling and Performance Optimization

- 11.1 Strategies for Scaling Node.js Applications.
- 11.2 Performance Optimization Techniques.
- 11.3 Load Balancing and Clustering.

Module – 12 Deployment and Hosting

- 12.1 Preparing Node.js Applications for Deployment.
- 12.2 Hosting options, Including Cloud Platforms.
- 12.3 Continuous Integration and Deployment (CI/CD).

Module – 13 Security Best Practices

- 13.1 Identifying and Mitigating Common Security Threats.
- 13.2 Implementing Security Measures in Node.js.
- 13.3 Handling Authentication Vulnerabilities.

Module – 14 Debugging and Testing

- 14.1 Debugging Node.js Applications.
- 14.2 Unit Testing and Integration Testing.
- 14.3 Tools and Best Practices for Testing.

Module – 15 Node.js Ecosystem and Trends

- 15.1 Exploring the Node.js Ecosystem and Community.
- 15.2 Trends and Emerging Technologies in Node.js.
- 15.3 Staying up-to-date with Node.js Developments.

1

Introduction to Node.js

1.1 What is Node.js…???

Node.js is an open-source, server-side JavaScript runtime environment built on the V8 JavaScript engine developed by Google. It allows developers to execute JavaScript code on the server, rather than in a web browser. This enables you to build scalable and efficient network applications, both for the web and beyond. Node.js was created by Ryan Dahl in 2009 and has gained significant popularity since then.

1.2 Features and Advantages of Node.js…?

1. Non-Blocking, Asynchronous I/O

Node.js is designed around an event-driven, non-blocking I/O model. This means that it can handle a large number of concurrent connections without the need for multi-threading, making it highly efficient. Instead of waiting for operations to complete, Node.js allows you to define callbacks, which are executed once the operation is finished.

- **javascript**

```
const fs = require('fs');
```

```
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
  } else {
    console.log(data);
  }
});
```

In the code above, “fs.readFile” does not block the execution of other code. It executes the callback function once the file is read.

2. Fast Execution

Node.js uses the V8 JavaScript engine, which compiles JavaScript into machine code, making it extremely fast. This is one of the reasons why Node.js is often used for building high-performance applications.

3. Single Programming Language

Using JavaScript for both the client-side and server-side allows developers to write full-stack applications using a single programming language. This can simplify the development process and reuse code between the client and server.

4. Rich Ecosystem of Packages

Node.js has a vast ecosystem of open-source packages available through npm (Node Package Manager). You can easily integrate these packages into your projects, saving you time and effort in building various functionalities.

5. Scalability

Node.js is designed to be highly scalable. It can handle a large number of concurrent connections efficiently, making it suitable for real-time applications like chat applications, online gaming, and more.

6. Active Community

Node.js has a large and active community of developers and contributors. This means that you can find extensive resources, libraries, and solutions to common problems.

1.3 Installation and Setup of Node.js

To get started with Node.js, you need to install it on your system. Here's how you can do it:

Download Node.js: Visit the official Node.js website (https://nodejs.org/) and download the installer for your operating system.

Install Node.js: Run the installer and follow the installation instructions. Node.js comes with npm (Node Package Manager), which is also installed during the process.

Verify Installation: Open your terminal or command prompt and type the following commands to verify that Node.js and npm are installed:

```
node -v
npm -v
```

You should see the versions of Node.js and npm printed to the console.

1.4 Your First Node.js “Hello World” Program

Now that you have Node.js installed, let's create a simple "Hello World" program to get a hands-on experience.

Create a new directory for your project and navigate to it using the terminal or command prompt.

Create a new file named hello.js and open it in a code editor.

Add the following code to hello.js:

- **javascript**

```
console.log("Hello, Node.js!");
```

Save the file.

In your terminal or command prompt, navigate to the directory where hello.js is located.

Run the program by executing the following command:

```
node hello.js
```

You should see "Hello, Node.js!" printed to the console.

Congratulations, you've just created and executed your first Node.js program!

2

Node.js Modules and NPM

2.1 Understanding Node.js Modules

In Node.js, modules are a fundamental concept that helps organize and structure your code. Modules are essentially reusable blocks of code that can include functions, objects, or variables. They allow you to encapsulate and separate functionality, making your code more maintainable and easier to work with.

Node.js provides a built-in module system, which means that you can use existing modules, create your own custom modules, and manage dependencies between them. Here's an overview of Node.js modules:

Core Modules:

Node.js has a set of built-in core modules like fs (file system), http (HTTP server), and path (file path utilities). You can use these modules without needing to install them separately.

- **javascript**

```
const fs = require('fs');

const http = require('http');
```

Custom Modules:

You can create your own modules to encapsulate code that performs specific tasks. To create a custom module, create a JavaScript file and export the functions, objects, or variables you want to make available to other parts of your application.

- **javascript**

```
// mymodule.js

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


module.exports = {
  greet,
};
```

You can then import and use this module in other parts of your application.

- **javascript**

```
const myModule = require('./mymodule');


console.log(myModule.greet('John')); // Output: Hello, John!
```

Third-Party Modules:

You can also use third-party modules created by the community or other developers. These modules are available through the Node Package Manager (NPM), which we'll discuss in the next section.

2.2 Creating and Using Custom Modules

Creating custom modules is a fundamental aspect of Node.js development. To create and use a custom module, follow these steps:

Create a JavaScript file that will serve as your module. For example, create a file named myModule.js.

Define the functionality within your module. Export functions, objects, or variables that you want to make available to other parts of your application using module.exports. Here's an example:

- **javascript**

```
// myModule.js
function greet(name) {
  return `Hello, ${name}!`;
}

module.exports = {
```

```
  greet,
};
```

In another file (e.g., your main application file), require the custom module using require. This allows you to use the exported functionality from your module.

- **javascript**

```
const myModule = require('./mymodule');

console.log(myModule.greet('Pawan')); // Output: Hello, Pawan!
```

Custom modules are a powerful way to organize your code into reusable components and keep your application modular and maintainable.

2.3 Introduction to Node.js Package Manager

NPM, which stands for Node Package Manager, is the default package manager for Node.js. It's a command-line tool that allows you to discover, install, and manage packages (i.e., libraries and modules) for your Node.js applications. NPM also comes pre-installed with Node.js.

NPM is an essential tool for Node.js development for several reasons:

Package Management: NPM simplifies the process of installing, updating, and removing packages for your projects.

Version Control: It allows you to specify the version of a package your project depends on, ensuring consistent behaviour.

Dependency Resolution: NPM can automatically resolve and install the dependencies of packages you use in your project.

Global and Local Packages: You can install packages globally (accessible from the command line) or locally (associated with a specific project).

Publishing Packages: NPM also allows developers to publish their own packages for others to use.

2.4 Install and Managing Packages with NPM

Here are the key NPM commands and their usage:

Initialize a Project: To start a new Node.js project, navigate to your project directory in the terminal and run:

```
npm init
```

Follow the prompts to create a package.json file, which will store information about your project and its dependencies.

Install Packages Locally: You can install packages locally in your project using:

```
npm install package-name
```

For example:

```
npm install lodash
```

This adds the lodash package to your project's node_modules directory and updates the package.json file to include the dependency.

Install Packages Globally: Some packages are meant to be used from the command line, and you can install them globally using the -g flag:

```
npm install -g package-name
```

For example:

```
npm install -g nodemon
```

This makes the package globally available as a command-line tool.

Save Packages as Dev Dependencies: You can save packages as development dependencies using the --save-dev or -D flag. These packages are typically used during development, but not in the production version of your application.

```
npm install --save-dev package-name
```

Uninstall Packages: To remove a package from your project, use the uninstall command:

```
npm uninstall package-name
```

Update Packages: To update packages in your project, use the update command:

```
npm update package-name
```

This will update the package to the latest version allowed by the version range defined in your package.json.

List Installed Packages: To list the packages installed in your project, use the “ls” or list command:

```
npm ls
```

This command displays a tree-like structure of your project's dependencies.

NPM is a powerful tool for managing dependencies and simplifying the process of integrating third-party libraries and modules into your Node.js projects. It plays an important role in the Node.js ecosystem, enabling developers to build applications more efficiently and maintainable by leveraging a vast library of open-source packages.

3

Asynchronous JavaScript

3.1 Asynchronous Programming in Node.js

In Node.js, asynchronous programming is crucial for tasks like handling I/O operations, network requests, and other operations that may take some time to complete. Instead of blocking the program's execution while waiting for these tasks to finish, Node.js uses an event-driven, non-blocking model.

Asynchronous Tasks in Node.js:

- Reading and writing files.
- Making HTTP requests.
- Database operations.
- Timers and scheduled events.
- Handling user input and interactions.

3.2 Callbacks and the Event Loop

Callbacks are functions that are passed as arguments to other functions and are executed at a later time. They are a common way to handle asynchronous operations in Node.js. The event loop manages the execution of these callbacks.

Example 1: Using Callbacks

- **javascript**

```
function fetchUserData(userId, callback) {
 // Simulate fetching user data (e.g., from a database)
 setTimeout(() => {
  const user = { id: userId, name: "John" };
  callback(user);
 }, 1000);
}

fetchUserData(123, (user) => {
 console.log(`User ID: ${user.id}, Name: ${user.name}`);
});
```

In this example, fetchUserData takes a callback function as an argument and simulates an asynchronous operation. When the operation is complete, the callback is executed.

The Event Loop

Node.js has an event loop that constantly checks the message queue for tasks. When an asynchronous task completes, it places a message in the queue. The event loop processes these messages and executes the associated callbacks.

3.3 Promises and Async/Await For Better….

While callbacks are useful, they can lead to callback hell or pyramid of doom when dealing with complex asynchronous operations. Promises and async/await were introduced to address this issue.

Promises provide a more structured way to handle asynchronous tasks and avoid deeply nested callbacks.

Example 2: Using Promises

- **javascript**

```
function fetchUserData(userId) {
  return new Promise((resolve, reject) => {
    // Simulate fetching user data
    setTimeout(() => {
      const user = { id: userId, name: "Alice" };
      resolve(user);
    }, 1000);
  });
}

fetchUserData(456)
```

```
  .then((user) => {
    console.log(`User ID: ${user.id}, Name: ${user.name}`);
  })
  .catch((error) => {
    console.error(error);
  });
```

In this example, fetchUserData returns a promise that resolves with the user data. We can use .then() to handle the resolved value and .catch() to handle errors.

Async/Await is built on top of promises and provides a more readable and synchronous-like way to write asynchronous code.

Example 3: Using Async/Await

- **javascript**

```
async function getUserInfo(userId) {
  try {
    const user = await fetchUserData(userId);
    console.log(`User ID: ${user.id}, Name: ${user.name}`);
  } catch (error) {
    console.error(error);
  }
```

```
}


getUserInfo(789);
```

Async functions, marked by the async keyword, can await promises, making the code appear more linear and easier to follow. The try...catch block handles errors.

4

File System and Streams

4.1 Working with File System in Node.js

Node.js provides a built-in module called fs (File System) that allows you to interact with the file system. You can perform various file operations, such as reading, writing, updating, and deleting files. Here's an overview of some common operations:

Reading Files

To read a file using the fs module, you can use the “fs.readFile” method. This method reads the entire content of a file into a buffer or string, depending on the encoding.

- **javascript**

```
const fs = require('fs');


fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
  } else {
```

```
  console.log(data);
 }
});
```

In this example, we use fs.readFile to read the contents of the 'example.txt' file. The second argument specifies the encoding, which is set to 'utf8' to read the data as a string.

Writing Files

To write data to a file, you can use the fs.writeFile method. It allows you to specify the file to write to and the content to be written.

- **javascript**

```
const fs = require('fs');


fs.writeFile('output.txt', 'Hello, Node.js!', (err) => {
 if (err) {
  console.error(err);
 } else {
  console.log('File written successfully.');
 }
});
```

In this example, we use fs.writeFile to create or overwrite the 'output.txt' file with the specified content.

Other File Operations

The “fs” module provides various other methods for file operations, including:

- **fs.appendFile:** Appends data to an existing file.
- **fs.rename:** Renames a file.
- **fs.unlink:** Deletes a file.
- **fs.stat:** Retrieves file information (e.g., size, permissions).
- **fs.mkdir and fs.rmdir:** Create and remove directories.

4.2 Reading and Writing Files

In Node.js, reading and writing files is a common task for a wide range of applications. Let's explore these operations in more detail.

Reading Files

To read the contents of a file, you can use the “fs.readFile” method as shown earlier. It's important to provide a callback function to handle the file data once it's read. Here's an example of reading a JSON file:

- **javascript**

```
const fs = require('fs');


fs.readFile('data.json', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
```

```
  } else {
    const jsonData = JSON.parse(data);
    console.log(jsonData);
  }
});
```

In this example, we read 'data.json', parse it as JSON, and then use the parsed data.

Writing Files

To write data to a file, you can use the “fs.writeFile” method. Here's an example of writing data to a file:

- **javascript**

```
const fs = require('fs');

const dataToWrite = 'This is the data to be written to the file.';
fs.writeFile('output.txt', dataToWrite, (err) => {
  if (err) {
    console.error(err);
  } else {
    console.log('Data written to the file.');
  }
});
```

In this example, we specify the data to be written and the file ('output.txt') to which the data will be written.

4.3 Using Streams for Efficient Data Processing

Working with files using the fs.readFile and fs.writeFile methods is suitable for small files, but it can be inefficient for large files or when you need to process data in real-time. Node.js offers a solution: streams.

What are Streams?

Streams are objects that allow you to read or write data piece by piece, rather than loading an entire file into memory. Streams are memory-efficient and enable you to work with data in a more responsive and performant way.

Node.js has several types of streams, including Readable, Writable, and Transform streams.

Reading Files with Streams

To read a file using streams, you can create a Readable stream and pipe it to a Writable stream to save the data.

- **javascript**

```
const fs = require('fs');
```

```
const readStream = fs.createReadStream('largeFile.txt');
const writeStream = fs.createWriteStream('output.txt');

readStream.pipe(writeStream);

readStream.on('end', () => {
  console.log('File read and written successfully.');
});

readStream.on('error', (err) => {
  console.error(err);
});

writeStream.on('finish', () => {
  console.log('Writing finished.');
});
```

In this example, we use createReadStream to read a large file ('largeFile.txt') and createWriteStream to create a new file ('output.txt') where the data will be written. We then pipe the data from the readable stream to the writable stream.

Writing Files with Streams

You can also use streams to write data to a file piece by piece. Here's an example of creating a writable stream to write data incrementally:

- **javascript**

```
const fs = require('fs');


const dataToWrite = 'This is a long text that will be written to the file using streams.';

const writeStream = fs.createWriteStream('output.txt');


writeStream.write(dataToWrite, 'utf8', () => {

  console.log('Data written to the file.');

  writeStream.end();

});


writeStream.on('finish', () => {

  console.log('Writing finished.');

});


writeStream.on('error', (err) => {

  console.error(err);

});
```

In this example, we create a writable stream and use the write method to write data in smaller chunks. The end method indicates the end of writing, and we handle events to determine when writing is finished or if there's an error.

Streams are particularly useful when working with large files, as they allow you to process data without loading it entirely into memory, resulting in better performance and efficiency.

5

HTTP and Web Servers

5.1 Creating HTTP Servers with Node.js

Node.js allows you to create HTTP servers effortlessly using the built-in http module. HTTP servers enable your Node.js applications to listen for incoming HTTP requests and respond accordingly. Below is a simple example of creating an HTTP server:

- **javascript**

```
const http = require('http');


const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, Node.js HTTP Server!');
});


const port = 3000;
server.listen(port, () => {
  console.log(`Server listening on port ${port}`);
});
```

Here's what's happening in this code:

- We require the http module to access its functionality.
- We create an HTTP server using the http.createServer method. It takes a callback function that will be executed whenever a request is received.
- Inside the callback, we set the response status code to 200 (OK) and specify the response content type as plain text.
- We use res.end() to send the response to the client.
- Finally, we listen on a specified port (e.g., 3000) and log a message when the server starts.

5.2 Handling HTTP Requests and Responses

Handling HTTP requests and responses is a fundamental aspect of building web applications. In the previous example, we've seen a simple response. However, a real web server should route requests to the appropriate handlers, allowing for dynamic and interactive behavior.

Routing Requests

To create more sophisticated web servers, you need to route requests to the appropriate handler functions. Here's an example using the http module's built-in routing capabilities:

- **javascript**

```
const http = require('http');
```

```
const server = http.createServer((req, res) => {
  if (req.url === '/') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Welcome to the homepage!');
  } else if (req.url === '/about') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('About us page');
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Page not found');
  }
});

const port = 3000;
server.listen(port, () => {
  console.log(`Server listening on port ${port}`);
});
```

In this example, we route requests based on the URL path, sending different responses for the homepage ('/'), the about page ('/about'), and any other path.

Serving Static Files

To serve static files like HTML, CSS, JavaScript, and images, you can use the fs (File System) module to read and send the file content in response to an HTTP request. Here's a simplified example serving an HTML file:

- **javascript**

```
const http = require('http');

const fs = require('fs');


const server = http.createServer((req, res) => {
  if (req.url === '/') {
    fs.readFile('index.html', (err, data) => {
      if (err) {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('File not found');
      } else {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.end(data);
      }
    });
  }
});
```

```
const port = 3000;

server.listen(port, () => {

  console.log(`Server listening on port ${port}`);

});
```

In this example, when a request is made to the root URL ('/'), we use fs.readFile to read the 'index.html' file and send its content as an HTML response.

5.3 Building- RESTful APIs with Node.js

REST (Representational State Transfer) is an architectural style for designing networked applications. RESTful APIs are a common way to expose data and services via HTTP. Building RESTful APIs is a vital part of web development, and Node.js is well-suited for this purpose.

To build a RESTful API with Node.js, you need to handle various HTTP methods (GET, POST, PUT, DELETE) and define routes and resources. You can use popular libraries like Express.js to simplify the process.

Example: Creating a Simple RESTful API with Express.js

First, you'll need to install the Express.js library:

```
npm install express
```

Now, you can create a simple RESTful API:

- **javascript**

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

// Sample data (in-memory database)
const todos = [
  { id: 1, text: 'Buy groceries' },
  { id: 2, text: 'Walk the dog' },
];

app.use(express.json());

// Get all todos
app.get('/todos', (req, res) => {
  res.json(todos);
});

// Get a specific todo by ID
app.get('/todos/:id', (req, res) => {
  const id = parseInt(req.params.id);
```

```
  const todo = todos.find((t) => t.id === id);

  if (todo) {
    res.json(todo);
  } else {
    res.status(404).json({ message: 'Todo not found' });
  }
});

// Create a new todo
app.post('/todos', (req, res) => {
  const newTodo = req.body;
  todos.push(newTodo);
  res.status(201).json(newTodo);
});

// Update a todo
app.put('/todos/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const updatedTodo = req.body;
  const index = todos.findIndex((t) => t.id === id);
```

```
  if (index !== -1) {
    todos[index] = { ...todos[index], ...updatedTodo };
    res.json(todos[index]);
  } else {
    res.status(404).json({ message: 'Todo not found' });
  }
});

// Delete a todo
app.delete('/todos/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const index = todos.findIndex((t) => t.id === id);

  if (index !== -1) {
    const deletedTodo = todos.splice(index, 1)[0];
    res.json(deletedTodo);
  } else {
    res.status(404).json({ message: 'Todo not found' });
  }
});
```

```
const port = 3000;

app.listen(port, () => {

  console.log(`RESTful API server listening on port ${port}`);

});
```

In this example, we use Express.js to create a RESTful API for managing a todo list. The API supports basic operations like retrieving all todos, getting a specific todo by ID, creating new todos, updatig todos, and deleting todos.

- We define routes and handle various HTTP methods (GET, POST, PUT, DELETE) for each route.
- We use in-memory storage (todos array) to simulate a database.
- We parse JSON request bodies using express.json() middleware.
- We handle each route's functionality, such as fetching, creating, updating, or deleting todos.
- We send appropriate HTTP responses based on the API's behaviour.

This is a simple example of a RESTful API, but real-world applications may require more features, validation, authentication, and database integration. Express.js is a highly flexible and extensible framework that can accommodate these requirements.

6

Express.js Framework

6.1 Introduction to Express.js

Express.js is a fast, unopinionated, and minimalist web application framework for Node.js. It simplifies the process of building web applications by providing a set of tools and features to handle common web-related tasks. Here are some key characteristics of Express.js:

Routing: Express provides a flexible and intuitive routing system that allows you to define how your application responds to different HTTP requests (GET, POST, PUT, DELETE, etc.).

Middleware: Middleware functions are at the core of Express.js. They allow you to perform various tasks such as request processing, authentication, and response generation in a modular and organized manner.

Template Engines: Express can be used with different template engines like EJS, Pug, and Handlebars to generate dynamic HTML pages on the server.

HTTP Utility Methods: It simplifies working with HTTP methods and request/response objects, making it easy to handle HTTP requests.

Static File Serving: Express makes it straightforward to serve static files like HTML, CSS, and JavaScript.

Error Handling: It provides built-in error handling mechanisms to streamline the handling of errors and exceptions.

Robust Community: With a large and active community, Express has a wealth of extensions and middleware packages available via npm.

6.2 Creating Web Application using Express.js

To get started with Express.js, you'll need to install it using npm. Here's how you can create a simple web application using Express:

Installation:

First, create a directory for your project and navigate to it in your terminal. Then, initialize a new Node.js project and install Express:

```
mkdir express-demo
cd express-demo
npm init -y
```

```
npm install express
```

Creating an Express Application:

Create an entry point file (e.g., app.js) and set up your Express application:

- **javascript**

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

const app = express();

const port = 3000;


// Define a route for the homepage

app.get('/', (req, res) => {

  res.send('Hello, Express.js!');

});


// Start the server

app.listen(port, () => {

  console.log(`Server is listening on port ${port}`);

});
```

Running the Application:

Start your Express application with the following command:

node app.js

You should see the message "Server is listening on port 3000" in the console. Open your web browser and navigate to http://localhost:3000 to see the "Hello, Express.js!" message.

This is a simple Express application that defines a single route for the homepage and sends a basic response.

6.3 Routing, Middleware, Template Engines

Express.js offers powerful features for building web applications, including routing, middleware, and template engines. Let's explore each of these in more detail.

Routing

Routing in Express.js allows you to define how your application responds to different HTTP requests based on the URL path and HTTP method (GET, POST, PUT, DELETE, etc.). Here's an example of defining routes in an Express application:

- **javascript**

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

const app = express();

// Define a route for the homepage
```

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

// Define a route for a specific product
app.get('/products/:id', (req, res) => {
  const productId = req.params.id;
  res.send(`Product ID: ${productId}`);
});

// Define a route for handling POST requests
app.post('/products', (req, res) => {
  res.send('Creating a new product');
});

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

In this example, we have defined routes for the homepage, a specific product page (using a parameter), and a route that handles POST requests for creating new products.

Middleware

Middleware functions in Express.js are functions that have access to the request (req) and response (res) objects and can perform tasks during the request-response cycle. Middleware functions can be added globally or applied to specific routes. Here's an example of using middleware for logging:

- **javascript**

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

const app = express();


// Custom middleware function for logging

function logMiddleware(req, res, next) {

  console.log(`Request received at ${new Date()}`);

  next(); // Continue to the next middleware or route

}


// Use the logMiddleware for all routes

app.use(logMiddleware);


// Define a route

app.get('/', (req, res) => {

  res.send('Homepage');

});
```

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

In this example, the logMiddleware function logs the time when a request is received. By using app.use(logMiddleware), we apply this middleware to all routes in the application.

Template Engines

Template engines allow you to generate dynamic HTML pages on the server. Express.js can work with various template engines such as EJS, Pug, and Handlebars. Here's an example using the EJS template engine:

First, install the EJS package:

```
npm install ejs
```

Next, configure Express to use EJS as the template engine:

- **javascript**

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

```
// Set EJS as the template engine
app.set('view engine', 'ejs');

// Define a route that renders an EJS template
app.get('/', (req, res) => {
  res.render('index', { title: 'Express.js', message: 'Hello, EJS!' });
});

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

Create an EJS template file named views/index.ejs:

- **html**

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
  </head>
  <body>
    <h1><%= message %></h1>
```

```
  </body>
</html>
```

In this example, we set EJS as the template engine using app.set('view engine', 'ejs') and render an EJS template in the route handler using res.render(). The template variables are defined in the second argument to res.render().

This is just the tip of the iceberg when it comes to Express.js. It offers a wide range of features and middleware for building web applications, including handling form data, authentication, sessions, and more. Express is widely used for creating RESTful APIs, web applications, and even full-fledged websites.