Express.js

Modules

1

Introduction to Express.js

1.1 Understanding the role of Express.js

Express.js, commonly known as Express, is a web application framework for Node.js. It is designed to simplify the process of building web applications and APIs by providing a robust set of features and tools. Express.js is built on top of Node.js, which is a server-side JavaScript runtime environment. It serves as a foundation for creating web servers and handling HTTP requests and responses effectively.

The Role of Express.js

Express.js plays a important role in web development by acting as an intermediary between the server and client. Its primary functions include:

1. Routing

Express allows you to define routes for different URLs and HTTP methods. This enables you to specify how your application should respond to various requests. For example, you can create routes for handling user authentication, retrieving data from a database, or serving static files like HTML, CSS, and JavaScript.

2. Middleware

Middleware functions are a core concept in Express. They are used to perform tasks such as request parsing, authentication, logging, and error handling. Middleware functions can be added to the request-response cycle, providing a way to modularize and customize the behavior of your application.

3. Templating

Express supports various templating engines like Pug, EJS, and Handlebars. These engines allow you to generate dynamic HTML pages by injecting data into templates. This is essential for rendering web pages with dynamic content, such as user profiles or product listings.

4. Static File Serving

Express simplifies the process of serving static files like images, stylesheets, and client-side JavaScript. You can define a directory where these files reside, and Express will automatically handle requests for them.

5. Middleware and Third-Party Packages

Express can be extended with a wide range of middleware and third-party packages available in the Node.js ecosystem. This extensibility allows you to add features like authentication with Passport.js, session management, and data validation with ease.

6. RESTful APIs

Express is an excellent choice for building RESTful APIs. It provides a clean and organized way to define API endpoints, handle request payloads, and send JSON responses, making it a popular framework for developing server-side components of web and mobile applications.

7. WebSocket Support

While primarily an HTTP server framework, Express can be integrated with WebSocket libraries like Socket.io to enable real-time communication between clients and servers.

1.2 Installation and Setup of Express.js

Before you can start using Express.js, you need to install it and set up a basic project structure. Follow these steps to get started:

Step 1: Install Node.js

Ensure that you have Node.js installed on your system. You can download the latest version from the official Node.js website (https://nodejs.org/).

Step 2: Create a New Directory for Your Project

Create a new directory where you want to work on your Express.js project. Open your terminal or command prompt and navigate to this directory.

```
mkdir my-express-app

cd my-express-app
```

Step 3: Initialize a Node.js Project

Run the following command to initialize a new Node.js project. This will create a `package.json` file, which will store information about your project and its dependencies.

```
npm init -y
```

Step 4: Install Express.js

To install Express.js, use npm (Node Package Manager) within your project directory:

```
npm install express
```

This command will download and install Express.js along with its dependencies into the “node_modules” directory of your project.

Step 5: Create an Express Application

Now that you have Express.js installed, you can create a basic Express application. Create a new JavaScript file (e.g., “app.js” or “index.js”) in your project directory.

- **javascript**

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

const app = express();

const port = 3000;


// Define a route

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

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

});


// Start the server

app.listen(port, () => {

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

});
```

In the code above:

- We import the Express.js module and create an instance of the Express application.
- We define a route that responds to HTTP GET requests at the root URL ("/") with the message "Hello, Express!".
- We start the server and listen on port 3000.

Step 6: Run Your Express Application

To run your Express application, execute the following command in your project directory:

```
node app.js
```

Your Express application will start, and you should see the message "Server is running on port 3000" in the console. You can then access your application by opening a web browser and navigating to “http://localhost:3000”.

Congratulations! You've successfully installed and set up a basic Express.js application.

1.3 Creating a Basic Express Application

In the code snippet provided in the previous section 1.2 (Create an Express Applications), we created a basic Express application that responds with "Hello, Express!" when accessed at the root URL. Let's break down the key components of this application:

Importing Express

We start by importing the Express.js module:

- **javascript**

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

This line allows us to use the functionalities provided by Express throughout our application.

Creating an Express Application

Next, we create an instance of the Express application:

- **javascript**

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

This “app” object represents our web application and provides methods to define routes, use middleware, and start the server.

Defining a Route

In the code snippet, we define a route using the “app.get()” method:

- **javascript**

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

Here's what happens in this code:

- “app.get('/')” specifies that we are defining a route for HTTP GET requests to the root URL ("/").
- The second argument is a callback function that takes two parameters, “req” and “res”. “req” represents the HTTP request, and “res” represents the HTTP response.
- Inside the callback function, we use “res.send()” to send the response "Hello, Express!" back to the client.

Starting the Server

Finally, we start the server and listen on a specified port (in this case, port 3000):

- **javascript**

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

The “app.listen()” method starts the server and listens on the specified port. When the server starts successfully, the callback function is executed, and a message is logged to the console.

You can customize this basic Express application by defining more routes, adding middleware, and integrating it with databases or other third-party packages as needed.

2
Routing and Middleware

In module 2, we will dive into two fundamental aspects of Express.js: Routing and Middleware. These are essential concepts that empower developers to create dynamic and efficient web applications.

2.1 Creating Routes - Handling HTTP Requests

Routing in Express.js

Routing is a core concept in Express.js that allows you to define how your application responds to different HTTP requests and URL paths. In Express, routes are defined using HTTP methods (such as GET, POST, PUT, DELETE) and URL patterns. Each route specifies a function to execute when a request matching that route is received.

Creating Basic Routes

Here's how to create basic routes in Express:

- **javascript**

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

const app = express();
```

```
// Define a route for GET requests to the root path

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

  res.send('This is the homepage');

});


// Define a route for POST requests to the “/submit” path

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

  res.send('Form submitted successfully');

});


// Define a route for all other paths

app.use((req, res) => {

  res.status(404).send('Page not found');

});


// Start the server

const port = 3000;

app.listen(port, () => {

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

});
```

In this example:

- We define a route for HTTP GET requests to the root path ('/'). When a user accesses the root URL, they receive the response 'This is the homepage.'
- We define a route for HTTP POST requests to the '/submit' path. This is often used for form submissions.
- We use a catch-all route (expressed as “app.use()”) to handle all other paths. If a user requests an undefined path, they receive a 'Page not found' response.
- The server is started on port 3000.

Dynamic Routes

Express also allows you to create dynamic routes using parameters in the URL. Parameters are indicated by a colon followed by the parameter name in the URL pattern. Here's an example:

- **javascript**

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID: ${userId}`);
});
```

In this example, the “:id” parameter is a placeholder for any value in the URL. When a user accesses a URL like '/users/123', the value '123' is extracted from the URL and made available in "req.params.id". You can then use this value to perform actions or look up data related to that user.

2.2 Defining Middleware Functions ….

Middleware functions are a powerful aspect of Express.js that allow you to add processing logic to incoming requests before they reach your route handlers. Middleware functions can perform tasks such as request parsing, authentication, logging, and error handling. They are executed in the order in which they are defined in your Express application.

Creating Middleware Functions

Here's how you can create and use middleware functions in Express:

- **javascript**

```
// Example middleware function
function logRequest(req, res, next) {
  console.log(`Received ${req.method} request for ${req.url}`);
  next(); // Call next() to pass control to the next middleware or route handler
}

// Using the middleware function
app.use(logRequest);
```

```
// Define a route that uses the middleware
app.get('/protected', (req, res) => {
  res.send('This route is protected');
});
```

In this example:

- We define a middleware function “logRequest” that logs information about incoming requests, such as the HTTP method and URL.
- The “next()” function is called to pass control to the next middleware or route handler. This is crucial to ensure that the request continues to be processed after the middleware logic is executed.
- We use “app.use()” to apply the “logRequest” middleware to all routes, meaning it will be executed for every incoming request.

Using Middleware for Authentication

Middleware functions are often used for implementing authentication in Express applications. Here's a simplified example:

- **javascript**

```
// Example middleware for authentication
function authenticate(req, res, next) {
  const isAuthenticated = /* Check if user is authenticated */;
  if (isAuthenticated) {
    next(); // Continue processing if authenticated
  } else {
    res.status(401).send('Unauthorized');
  }
}


// Apply authentication middleware to a specific route
app.get('/protected', authenticate, (req, res) => {
  res.send('This route is protected');
});
```

In this example:

- The “authenticate” middleware checks if the user is authenticated. If authenticated, it calls “next()” to allow the request to proceed; otherwise, it sends a 'Unauthorized' response with a status code of 401.
- We apply the “authenticate” middleware only to the '/protected' route, ensuring that it's executed only for that specific route.

2.3 Routing Parameters and Route Chaining

Routing Parameters

Express.js allows you to extract data from URL parameters, as shown earlier with dynamic routes. You can access these parameters using “req.params”. Here's a more detailed example:

- **javascript**

```
app.get('/users/:id/posts/:postId', (req, res) => {
  const userId = req.params.id;
  const postId = req.params.postId;
  res.send(`User ID: ${userId}, Post ID: ${postId}`);
});
```

In this example, we define a route that captures two parameters, “:id” and “:postId”, from the URL. These values are then accessed using “req.params”.

Route Chaining

Route chaining is a technique used to apply multiple route handlers to a single route. This is particularly useful for breaking down the handling of a route into smaller, reusable components.

- **javascript**

```javascript
// Middleware for authentication
function authenticate(req, res, next) {
  const isAuthenticated = /* Check if user is authenticated */;
  if (isAuthenticated) {
    next();
  } else {
    res.status(401).send('Unauthorized');
  }
}

// Middleware for logging
function logRequest(req, res, next) {
  console.log(`Received ${req.method} request for ${req.url}`);
  next();
}

// Define a route and chain the middleware
app.get(
  '/protected',
  authenticate,
  logRequest,
  (req, res) => {
    res.send('This route is protected');
  }
);
```

In this example:

- We define two middleware functions, “authenticate” and “logRequest”.
- We use route chaining by passing an array of middleware functions followed by the route handler function to “app.get()”. This ensures that both “authenticate” and “logRequest” middleware functions are executed before the final route handler.

Route chaining allows you to create modular and organized routes by breaking them down into smaller, reusable middleware components.

3

Templating Engines

In module 3, we will dive into the world of templating engines in Express.js. Templating engines enable you to generate dynamic HTML content by injecting data into templates.

3.1 Working with Template Engines…..

What are Templating Engines?

Templating engines are libraries or frameworks that help you create dynamic HTML content by combining templates and data. They provide a way to structure and organize your HTML templates while allowing you to inject dynamic data seamlessly. In Express.js, you can choose from various templating engines, with EJS (Embedded JavaScript) and Handlebars being popular choices.

EJS (Embedded JavaScript)

EJS is a templating engine that lets you embed JavaScript code directly within your HTML templates. This makes it easy to incorporate data and logic into your views. Here's a simple example of using EJS in Express:

- **javascript**

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

const app = express();

const port = 3000;


// Set EJS as the view engine

app.set('view engine', 'ejs');


// Define a route that renders an EJS template

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

  const data = { message: 'Hello, EJS!' };

  res.render('index', { data });

});


app.listen(port, () => {

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

});
```

In this example:

- We set EJS as the view engine using "app.set('view engine', 'ejs')".

- We define a route that renders an EJS template called 'index'. The template is located in a directory named 'views'.
- We pass data to the template using the “{ data }” object. This data can be accessed in the EJS template.

Handlebars

Handlebars is another popular templating engine for Express.js. It follows a more minimalistic syntax compared to EJS. Here's an example of using Handlebars in Express:

- **javascript**

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

const exphbs = require('express-handlebars');

const app = express();

const port = 3000;


// Set Handlebars as the view engine

app.engine('handlebars', exphbs());

app.set('view engine', 'handlebars');


// Define a route that renders a Handlebars template

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

  const data = { message: 'Hello, Handlebars!' };
  res.render('index', { data });
});

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

In this example:

- We use the “express-handlebars” package to integrate Handlebars with Express.js.
- Handlebars templates are stored in a directory named 'views' by default.
- We pass data to the template, which can be accessed using Handlebars syntax.

3.2 Rendering Dynamic Views and Templates

Now that we have set up our preferred templating engine, let's explore how to render dynamic views and templates in Express.js.

Rendering Views

In Express.js, you render views using the “res.render()” method. This method takes two arguments: the name of the view (without the file extension) and an optional object containing data to pass to the view.

- **javascript**

```
app.get('/', (req, res) => {
  const data = { message: 'Hello, Express!' };
  res.render('index', { data });
});
```

In this example, the 'index' view is rendered with the provided data. The templating engine processes the template, injects the data, and sends the resulting HTML to the client.

Using Data in Templates

Both EJS and Handlebars allow you to access and display data within your templates. Here's how you can do it in each:

EJS Example:

- **html**

```
<!-- views/index.ejs -->
<!DOCTYPE html>
<html>
<head>
```

```
  <title>Express EJS Example</title>
</head>
<body>
  <h1><%= data.message %></h1>
</body>
</html>
```

In EJS, you use “<%= ... %>” to embed JavaScript code and output data within your HTML template. In this case, “data.message” is displayed as an “<h1>” heading.

Handlebars Example:

- **html**

```
<!-- views/index.handlebars -->
<!DOCTYPE html>
<html>
<head>
  <title>Express Handlebars Example</title>
</head>
<body>
  <h1>{{ data.message }}</h1>
</body>
```

```
</html>
```

In Handlebars, you use “{{ ... }}” to display data. Here, “data.message” is displayed as an “<h1>” heading.

Both EJS and Handlebars offer additional features for controlling the flow of your templates, including conditionals, loops, and partials (reusable template components).

3.3 Passing Data to Views from Express

In Express.js, you can pass data to views from your route handlers, allowing you to dynamically generate content based on server-side data. We've already seen examples of this in previous sections, but let's dive deeper into how to pass data to views.

Data Object

To pass data to a view, you create a JavaScript object containing the data you want to make available in the view. This object is then passed as the second argument to “res.render()”.

- **javascript**

```
app.get('/', (req, res) => {
  const data = { message: 'Hello, Express!' };
  res.render('index', { data });
});
```

In this example, the “data” object contains a single key-value pair, where the key is 'message' and the value is 'Hello, Express!'. This data can be accessed in the view using the templating engine's syntax.

Dynamic Data

In practice, data passed to views is often generated dynamically based on user requests, database queries, or other server-side logic. For example, you might fetch user information from a database and pass it to a user profile view:

- **javascript**

```
app.get('/profile/:id', (req, res) => {
  const userId = req.params.id;
  // Fetch user data from the database based on userId
  const userData = /* Database query logic */;
  res.render('profile', { user: userData });
});
```

In this example, the “userData” object contains user-specific data fetched from the database. This data is then passed to the 'profile' view, where it can be used to render the user's profile page.

Conditional Rendering

One common use case for passing data is to conditionally render content based on the data. For instance, you might want to display a different message to logged-in and logged-out users:

- **javascript**

```
app.get('/', (req, res) => {
  const isAuthenticated = /* Check if user is authenticated */;
  const message = isAuthenticated ? 'Welcome, User!' : 'Please log in.';
  res.render('index', { message });
});
```

In this example, the “isAuthenticated” variable is used to determine whether the user is logged in. Based on this condition, a different message is passed to the 'index' view, which is then displayed accordingly.

4

Handling Forms and Data

In module 4, we will dive into the essential aspects of handling forms and data in Express.js. Express.js provides tools and middleware to effectively parse, validate, and process form data, query parameters, and request bodies.

4.1 Parsing and Handling form Data in Express

What is Form Data?

Form data refers to the information submitted by users through HTML forms on web pages. This data can include various types of inputs such as text fields, checkboxes, radio buttons, and file uploads. Handling this data on the server side is essential for processing user actions, such as user registration, login, search queries, and more.

Handling Form Data with Express

Express.js simplifies the process of handling form data by providing middleware for parsing incoming requests. Two common middleware options for handling form data are `body-parser` and the built-in “express.urlencoded()”.

Using “body-parser” Middleware

The “body-parser” middleware is a popular choice for parsing form data in Express.js applications. To use it, you need to install the “body-parser” package and include it in your Express application.

- **javascript**

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const port = 3000;

// Use bodyParser middleware to parse form data
app.use(bodyParser.urlencoded({ extended: false }));

// Define a route to handle a form submission
app.post('/submit', (req, res) => {
  const formData = req.body;
  // Process the form data here
  res.send(`Form submitted: ${JSON.stringify(formData)}`);
});

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

In this example:

- We include the “body-parser” middleware using “app.use()”.
- The “bodyParser.urlencoded()” middleware is used to parse form data from incoming POST requests.
- When a form is submitted, the data is available in “req.body”. You can then process and respond to the data as needed.

Using “express.urlencoded()” Middleware (Express 4.16+)

Starting from Express 4.16.0, you can use the built-in “express.urlencoded()” middleware to parse form data without installing “body-parser”. This middleware is included by default in Express.

- **javascript**

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

const port = 3000;


// Use express.urlencoded() middleware to parse form data

app.use(express.urlencoded({ extended: false }));


// Define a route to handle a form submission

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

  const formData = req.body;

  // Process the form data here

  res.send(`Form submitted: ${JSON.stringify(formData)}`);

});


app.listen(port, () => {

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

});
```

In this example, we use “express.urlencoded()” to parse form data, which is a convenient option for modern versions of Express.

4.2 Working with Query Parameters – Req...

In Express.js, you can access data submitted through forms using both query parameters and request bodies. Query parameters are typically used with HTTP GET requests, while request bodies are used with HTTP POST requests (and other methods).

Query Parameters

Query parameters are key-value pairs included in the URL of an HTTP request. They are often used for filtering, sorting, or specifying additional data. To access query parameters in Express, you can use “req.query”.

- **javascript**

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


const app = express();

const port = 3000;


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

  const searchTerm = req.query.q;

  // Use searchTerm to perform a search

  res.send(`Searching for: ${searchTerm}`);

});
```

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

In this example, a user can access the '/search' route with a query parameter 'q' to specify a search term. The value of 'q' is then accessed using “req.query.q”.

Request Bodies

Request bodies are used to send data to the server in the body of an HTTP request, typically with POST requests. To access request bodies in Express, you can use “req.body” after parsing the body data.

- **javascript**

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const port = 3000;

app.use(bodyParser.urlencoded({ extended: false }));
app.post('/submit', (req, res) => {
  const formData = req.body;
  // Process the form data here
  res.send(`Form submitted: ${JSON.stringify(formData)}`);
});

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

In this example, when a form is submitted to the '/submit' route using a POST request, the data is available in “req.body” after parsing with “body-parser”. You can then process and respond to the data as needed.

4.3 Validating and Sanitizing user Inputs

Handling user inputs is not just about retrieving data but also about ensuring its validity and security. Users can submit malicious or incorrect data, which can lead to security vulnerabilities and application errors. Express.js provides mechanisms for validating and sanitizing user inputs to mitigate these risks.

Validation

Validation is the process of checking if user input meets certain criteria or constraints. Express.js doesn't include built-in validation libraries, but you can use third-party packages like `express-validator` or implement custom validation logic.

Using “express-validator” (Example)

The `express-validator` package simplifies input validation and sanitization in Express.js applications. To use it, you need to install the package and configure your routes to validate user input.

- **javascript**

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

const { body, validationResult } = require('express-validator');


const app = express();

const port = 3000;


app.use(express.json()); // Enable JSON parsing


// Define a route with input validation

app.post(

  '/submit',
  [
    // Validate the 'email' field
    body('email').isEmail(),
    // Validate the 'password' field
    body('password').isLength({ min: 6 }),
  ],
  (req, res) => {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    // Process the validated form data here
    res.json({ message: 'Form submitted successfully' });
  }
);

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

In this example:

- We use “express-validator” to define validation rules for the 'email' and 'password' fields.
- The “validationResult” function checks if there are validation errors in the request.
- If validation fails, a response with a 400 status code and error details is sent to the client.

Sanitization

Sanitization is the process of cleaning and modifying user input to remove or neutralize potentially harmful content. This is crucial for protecting your application from security vulnerabilities like Cross-Site Scripting (XSS) attacks.

Sanitizing User Input (Example)

You can use a package like “express-validator” to perform sanitization as well. Here's an example of sanitizing user input:

- **javascript**

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

const { body, validationResult } = require('express-validator');


const app = express();
const port = 3000;

app.use(express.json()); // Enable JSON parsing

// Define a route with input sanitization
app.post(
  '/submit',
  [
    // Sanitize the 'email' field
    body('email').trim().normalizeEmail(),
  ],
  (req, res) => {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    // Process the sanitized form data here
    res.json({ message: 'Form submitted successfully' });
  }
);


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

In this example:

- We use “express-validator” to sanitize the 'email' field by removing whitespace and normalizing the email address.
- The sanitized input is then available for further processing, reducing the risk of security vulnerabilities.

5

Structuring Express Applications

In module 5, we will dive into the best practices and techniques for structuring Express.js applications. Proper organization and project structure are essential for building scalable, maintainable, and readable applications.

5.1 Organizing Code and Project Structure

Why Structure Matters

Proper organization and project structure are critical for the long-term maintainability and scalability of your Express.js applications. A well-structured application is easier to understand, modify, and extend, making it more manageable as your project grows.

Common Project Structure

While there isn't a one-size-fits-all structure for Express.js applications, many developers follow common patterns and best practices. Here's a typical project structure for an Express app:

```
my-express-app/
 ├── node_modules/
 ├── public/
 |   ├── css/
 |   ├── js/
 |   └── images/
 ├── routes/
 |   ├── index.js
 |   ├── users.js
 |   └── ...
 ├── controllers/
 |   ├── indexController.js
 |   ├── usersController.js
 |   └── ...
 ├── views/
 |   ├── index.ejs
 |   ├── user.ejs
 |   └── ...
 ├── app.js
 ├── package.json
 └── ...
```

In this structure:

- **“node_modules”:** Contains project dependencies.
- **“public”:** Holds static assets like CSS, JavaScript, and images.
- **“routes”:** Defines route handling logic and route-specific middleware.
- **“controllers”:** Contains controller functions responsible for handling route logic.
- **“views”:** Stores template files used for rendering HTML pages.
- **“app.js”:** The main application file where Express is initialized and configured.
- **“package.json”:** Contains project metadata and dependencies.

5.2 Separating Routes and Controllers

Separation of Concerns

One of the fundamental principles of software design is the separation of concerns. In the context of an Express.js application, this means separating the routing logic (how requests are handled) from the business logic (what happens when a request is received).

Routing in Express

Routing defines how an application responds to client requests. In Express, you can define routes in the `routes` directory or directly in the main “app.js” file. However, it's recommended to organize routes into separate files.

Example of Routing

- **javascript**

// routes/index.js

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

router.get('/', (req, res) => {
  res.render('index');
});

module.exports = router;
```

In this example, we define a route for the root URL ('/') in the “index.js” file within the “routes” directory. This route responds by rendering an 'index' view.

Controllers in Express

Controllers are responsible for handling the business logic associated with specific routes. They should contain functions that perform actions related to the route, such as processing data, interacting with databases, and sending responses.

Example of a Controller

- **javascript**

```
// controllers/indexController.js

const indexController = {};

indexController.renderIndex = (req, res) => {
  res.render('index');
};

module.exports = indexController;
```

In this example, we create an “indexController” object with a function “renderIndex”. This function renders the 'index' view.

Connecting Routes and Controllers

To connect routes with controllers, you can require the controller module in your route files and invoke the relevant controller functions when defining routes.

Connecting Route and Controller

- **javascript**

```
// routes/index.js
```

```
const express = require('express');
const router = express.Router();
const indexController = require('../controllers/indexController');

router.get('/', indexController.renderIndex);

module.exports = router;
```

In this example, we import the “indexController” and use its “renderIndex” function as the route handler for the root URL ('/').

5.3 Best Practices for Structuring Express Apps

While the project structure and organization may vary depending on your specific requirements, adhering to best practices can help maintain a clean and efficient Express.js application.

1. Use Express Generator

If you're starting a new Express project, consider using the [Express Generator](https://expressjs.com/en/starter/generator.html). It provides a basic project structure with sensible defaults, including routes, controllers, and views.

2. Separate Concerns

Adhere to the separation of concerns principle. Keep your routing and controller logic separate to enhance code readability and maintainability.

3. Modularize Your Code

Split your application into modular components, such as routes, controllers, and middleware. Organize them in separate files and directories to make the codebase more manageable.

4. Choose a Consistent Naming Convention

Follow a consistent naming convention for your files, routes, and controllers. This makes it easier to locate and identify components within your project.

5. Use Middleware Wisely

Leverage middleware for common tasks like authentication, logging, and error handling. Keep middleware functions organized and avoid duplicating code.

6. Error Handling

Implement centralized error handling. You can use Express's built-in error-handling middleware or create custom error handling to centralize error management and provide consistent error responses.

- **javascript**

```
// Error handling middleware
app.use((err, req, res, next) => {
  // Handle errors here
  res.status(err.status || 500).send('Something went wrong');
});
```

7. Maintain a Clean “app.js”

Keep your “app.js” or main application file clean and focused on configuration and setup. Place your routes, controllers, and other components in separate files and require them in your main file.

8. Versioning Your API

If you're building a RESTful API, consider versioning your endpoints from the beginning. This allows you to make changes and updates to your API without breaking existing clients.

9. Documentation

Document your code, especially if you're working on a team or open-source project. Use comments and README files to explain the purpose and usage of different components.

10. Testing

Implement testing for your Express application. Tools like Mocha, Chai, and Supertest can help you write and run tests to ensure your application functions correctly.

11. Use an ORM/ODM

If your application interacts with a database, consider using an Object-Relational Mapping (ORM) or Object-Document Mapping (ODM) library like Sequelize, Mongoose, or TypeORM to simplify database operations and structure.

12. Keep Security in Mind

Prioritize security by validating and sanitizing user inputs, using HTTPS, implementing proper authentication, and following security best practices.

6

Authentication and Authorization

In module 6, we will dive into the aspects of user authentication and authorization within Express.js applications. These functionalities are vital for building secure and controlled access to your web applications.

6.1 Implementing...........

What is User Authentication?

User authentication is the process of verifying the identity of a user, typically by requiring them to provide credentials like a username and password. It ensures that users are who they claim to be before granting access to protected resources.

In Express.js, user authentication is often implemented using middleware, libraries, or custom logic.

Session Management

Session management is the practice of creating and managing sessions for authenticated users. A session represents a period of interaction between a user and a web application. It allows the application to store and retrieve user-specific data between requests.

Express.js provides mechanisms for handling session management, with the most commonly used library being “express-session”.

Using “express-session”

To use “express-session”, you first need to install it and set it up in your Express application.

- **javascript**

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

const app = express();
const port = 3000;

// Configure express-session middleware
app.use(
  session({
    secret: 'your_secret_key',
    resave: false,
    saveUninitialized: true,
  })
);

// Define a route that sets a session variable
app.get('/set-session', (req, res) => {
  req.session.username = 'john.doe';
  res.send('Session variable set');
});

// Define a route that reads the session variable
app.get('/get-session', (req, res) => {
  const username = req.session.username;
  res.send(`Session username: ${username}`);
});

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

In this example:

- We configure the “express-session” middleware and set a secret key for session encryption.

- A session variable “username” is set in the '/set-session' route.
- The '/get-session' route reads the session variable and responds with the username.

6.2 Handling User Registration and Login

User Registration

User registration is the process of allowing users to create accounts in your application. When a user registers, their credentials are stored securely in a database.

Here's an overview of the steps involved in user registration:

1. Collect user information, including username and password.
2. Validate and sanitize user inputs to prevent malicious data.
3. Hash and salt the password before storing it in the database.
4. Create a new user record in the database.

Here's an example using the popular “bcrypt” library for password hashing:

- **javascript**

```
const express = require('express');
const bodyParser = require('body-parser');
const bcrypt = require('bcrypt');

const app = express();
const port = 3000;

app.use(bodyParser.urlencoded({ extended: false }));

// In-memory database for demonstration (use a real database in production)
const users = [];

// Register a new user
app.post('/register', async (req, res) => {
  const { username, password } = req.body;

  // Check if the username already exists
  if (users.some((user) => user.username === username)) {
    return res.status(400).send('Username already exists');
  }
  // Hash and salt the password
  const saltRounds = 10;
  const hashedPassword = await bcrypt.hash(password, saltRounds);

  // Create a new user record
  users.push({ username, password: hashedPassword });

  res.send('Registration successful');
});

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

In this example:

- We collect the username and password from the registration form.
- Check if the username is already in use to prevent duplicate accounts.
- Use “bcrypt” to hash and salt the password before storing it in memory (replace with a real database in production).

User Login

User login is the process of verifying a user's credentials when they attempt to access their account. Users provide their username and password, which are checked against stored credentials.

Here's an overview of the steps involved in user login:

1. Collect user-provided username and password.
2. Retrieve the stored hashed password for the given username.
3. Compare the hashed password with the provided password.
4. If they match, the user is authenticated and can access their account.

Here's an example of user login:

- **javascript**

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

const bodyParser = require('body-parser');

const bcrypt = require('bcrypt');


const app = express();

const port = 3000;
```

```
app.use(bodyParser.urlencoded({ extended: false }));

// In-memory database for demonstration (use a real database in production)
const users = [];

// Login route
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  // Find the user by username
  const user = users.find((user) => user.username === username);

  // User not found
  if (!user) {
    return res.status(401).send('Invalid username or password');
  }

  // Compare the provided password with the stored hashed password
 const match = await bcrypt.compare(password, user.password);
  if (!match) {
    return res.status(401).send('Invalid username or password');
  }

  res.send('Login successful');
});

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

In this example:

- We collect the username and password provided during login.
- We retrieve the user's stored hashed password based on the username.
- We use “bcrypt” to compare the provided password with the stored hashed password. If they match, the user is authenticated.

6.3 Role-Based Access Control and ……..

Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) is a security model that defines access permissions based on user roles. In an Express.js application, RBAC helps you control who can perform specific actions or access certain resources.

To implement RBAC, you typically:

1. Define roles, such as 'admin', 'user', 'guest', etc.
2. Assign roles to users during registration or through an administrative interface.
3. Define authorization middleware that checks if a user has the required role to access a route or resource.

Here's a simplified example of RBAC using middleware in Express:

- **javascript**

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

const app = express();
const port = 3000;

// Mock user with roles (replace with real user data)
const user = {
  username: 'john.doe',
  roles: ['user', 'admin'],
};

// Authorization middleware
function authorize(roles) {
  return (req, res, next) => {
    if (roles.includes(user.roles)) {
      next(); // User has the required role
    } else {
      res.status(403).send('Access denied');
    }
  };
}

// Protected route accessible only to users with 'admin' role
app.get('/admin-panel', authorize(['admin']), (req, res) => {
  res.send('Welcome to the admin panel');
});

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

In this example:

- We define a \`user\` object with roles.
- The \`authorize\` middleware checks if the user has the required role to access the '/admin-panel' route.
- If the user has the 'admin' role, they can access the route; otherwise, access is denied.

Real-World RBAC

In a real-world application, RBAC often involves more complex structures, such as managing roles and permissions in a database, defining granular permissions, and handling role changes dynamically. You may also use third-party libraries or frameworks like Passport.js or Auth0 for more advanced authentication and authorization features.