Express

Module 5

ADC502 : Web Development

Introduction

What is Express

- Express.js, commonly referred to as Express, is a popular and widely used web application framework for Node.js.
- Express can be assumed as a layer built on the top of Node.js that helps manage a server and routes.
- Express is designed to make it easier to build web applications and APIs (Application Programming Interfaces) by providing a set of features and tools that simplify common web development tasks.

Advantages of Express.js

- **Simplicity**: Express has a minimalist and straightforward API, making it easy for developers to learn and use. It doesn't add unnecessary complexity, which is perfect for building small to large applications.
- **Middleware**: The middleware system allows developers to easily add, remove, and order components in the request/response pipeline, making it highly customizable and enabling the use of third-party middleware.
- **Routing**: Express provides a clean and organized way for developers to define routes for their applications, making it easy to handle various HTTP methods and URL patterns.
- **Performance**: Express is known for its speed and low overhead. It's built on Node.js, which is known for its non-blocking, event-driven architecture, making it efficient for handling a high volume of requests.

Advantages of Express.js

- **Community and Ecosystem**: Express enjoys the support of a large and active community, providing developers with a wealth of third-party middleware and extensions, simplifying the addition of functionality to their applications.
- **Flexibility**: Express doesn't enforce a strict project structure or dictate how developers should organize their code, allowing them to structure their applications according to their preferences and requirements.
- **Real-time Applications**: With the integration of WebSockets and modules like Socket.io, Express is well-suited for developers building real-time applications and chat systems.
- **Security**: While Express itself doesn't handle security, it equips developers with the tools and flexibility to integrate security measures such as authentication, authorization, and data validation to protect their applications.

Advantages of Express.js

- **Session and Cookie Handling**: Express simplifies the management of user sessions and cookies, making it easier for developers to implement user authentication and maintain state between requests.
- **Active Development**: The framework is actively developed and maintained, ensuring that developers can expect updates, bug fixes, and security patches, ensuring the longevity of their projects.
- **Scalability**: Express can be used to build both small and large applications, and its modular nature allows developers to add components and scale their applications as needed. When combined with load balancers, it can handle increased traffic and scale horizontally.
- **Cross-platform Compatibility**: Since Express is built on Node.js, it can run on various platforms, making it versatile for deploying applications to different hosting environments.

First App

Steps for Express “Hello World”

- Install Node.js
- Create a Project Directory:
  - *mkdir my-express-app*
  - *cd my-express-app*
- Initialize a Node.js Project:
  - *npm init*
- Install Express.js:
  - *npm install express --save*

Steps for Express “Hello World”

- Create an Express Application: Create a JavaScript file (e.g., app.js or index.js) in your project directory. This file will contain your Express application code.

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

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

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

Steps for Express “Hello World”

- Run Express Application:
  - *node app.js*

- Access Your Application: Open a web browser or use a tool like cURL or a REST client to access your application at **http://localhost:3000** (or the port you specified in your code). You should see "Hello, Express!" displayed in your browser.

Express Router

Express Router

- An "Express router" is a key component in the Express.js framework, used by developers to manage and organize the routing of web applications and APIs.
- It acts as a mini-application within the larger Express application, allowing developers to define and group related routes, middleware, and functions together.
- In simpler terms, think of it as a way to compartmentalize and structure the code for handling different routes and requests in an organized manner.
- With Express routers, developers can create separate sets of routes for various parts of their application, such as user authentication, data retrieval, or administrative functions.

Express Router

- To use an Express router, developers first create an instance of the router, define routes and their associated functions, and then attach the router to the main Express application.
- This separation of concerns simplifies the process of adding, modifying, or removing routes, making it an essential tool for structuring and organizing the routing logic in Express.js applications.

Advantages of Express Routers

- **Modularity**: They encourage a modular approach to application design, making the codebase easier to manage and maintain. Developers can split the routing logic into smaller, self-contained pieces.
- **Code Reusability**: Routers can be reused in multiple parts of the application or across different projects, reducing duplication of code.
- **Improved Readability**: By grouping related routes and their associated middleware functions, routers enhance the overall readability and maintainability of the code.
- **Scalability**: As the application grows, it's easier to add new routes or update existing ones within dedicated routers, without affecting other parts of the application.

Example

```
// Import necessary modules
const express = require('express');
const app = express();
const port = 3000;

// Create an instance of an Express router
const router = express.Router();
```

// Define a route on the router

```
router.get('/', (req, res) => {
  res.send('This is the main route');
});
```

Example (contd.)

```
// Define another route on the router
router.get('/about', (req, res) => {
  res.send('This is the about page');
});

// Attach the router to the main Express application
app.use('/app', router); // This means all routes defined on the router will start with '/app'

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

Example (contd.)

- In this example:
  - We create an Express application and set it to listen on port 3000.
  - We create an instance of an Express router using express.Router().
  - We define two routes on the router using .get(). One route responds to requests to the root path ("/") and the other to "/about."
  - We attach the router to the main Express application using app.use(). We specify that all routes defined on the router should start with "/app." For example, the route defined as "/about" on the router is accessed as "/app/about" in the application.
- Accessing the URLs:
  - http://localhost:3000/app/ will respond with "This is the main route."
  - http://localhost:3000/app/about will respond with "This is the about page."

REST API

What is RESTful API

- For theory refer this page from AWS
  https://aws.amazon.com/what-is/restful-api/

Examples of open RESTful APIs provided by popular websites and services:

- GitHub API:
  - GitHub offers an open REST API that allows developers to access and interact with GitHub data, such as repositories, issues, pull requests, and user profiles. It's widely used for integrating GitHub features into applications and automating workflows.
- Google Maps API:
  - Google Maps offers a suite of APIs for geolocation and mapping services. Developers can use these APIs to integrate maps, location data, and directions into their web and mobile applications.
- OpenWeatherMap API:
  - OpenWeatherMap provides weather data through its open REST API. Developers can access current weather conditions, forecasts, and historical weather data for locations around the world.
- YouTube Data API:
  - YouTube's API allows developers to interact with YouTube's vast video repository. Developers can search for videos, retrieve video metadata, upload videos, and manage user accounts.

REST API using Express

```
// Import necessary modules
const express = require('express');
const app = express();
const port = 3000;

// Sample data (in-memory)
const books = [
  { id: 1, title: 'Book 1' },
  { id: 2, title: 'Book 2' },
  { id: 3, title: 'Book 3' },
];
// Define a route to get all books
app.get('/api/books', (req, res) => {
  res.json(books);
});
```

```
// Define a route to get a specific book by ID
app.get('/api/books/:id', (req, res) => {
  const bookId = parseInt(req.params.id);
  const book = books.find((b) => b.id === bookId);

  if (!book) {
    return res.status(404).json({ error: 'Book not found' });
  }
  res.json(book);
});
// Start the server
app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

Access the API

- Access the REST API using tools like a web browser or cURL:
  - To get all books: http://localhost:3000/api/books
  - To get a specific book (e.g., ID 1): http://localhost:3000/api/books/1
- The API will return JSON responses with the list of books or a specific book, and it handles routes and HTTP methods according to REST principles.

Authentication

Middleware

- Middleware in Express.js serves as an intermediary layer that processes HTTP requests and responses within an Express application.
- It consists of functions that can be added to the request-response cycle to perform various tasks, such as logging, data validation, authentication, and more.
- Middleware functions are executed sequentially, and they have access to the request and response objects, allowing them to modify or augment the request and response data.

Example : Authentication using Middleware

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

// Sample user data (usually, this would come from a database)
const users = [
  { id: 1, username: 'user1', password: 'password1' },
  { id: 2, username: 'user2', password: 'password2' },
];
```

Example : Authentication using Middleware

```
// Middleware for user authentication
const authenticateUser = (req, res, next) => {
  // Extract credentials from the request (e.g., from request headers or cookies)
  const { username, password } = req.headers;

  // Find the user in the user array based on provided credentials
  const user = users.find((u) => u.username === username && u.password === password);
```

Example : Authentication using Middleware

```
if (user) {
    // User is authenticated, attach the user object to the request for later use
    req.user = user;
    next(); // Proceed to the next middleware or route handler
  } else {
    // User authentication failed, return an unauthorized response
    res.status(401).json({ error: 'Unauthorized' });
  }
}; // middleware over
```

Example : Authentication using Middleware

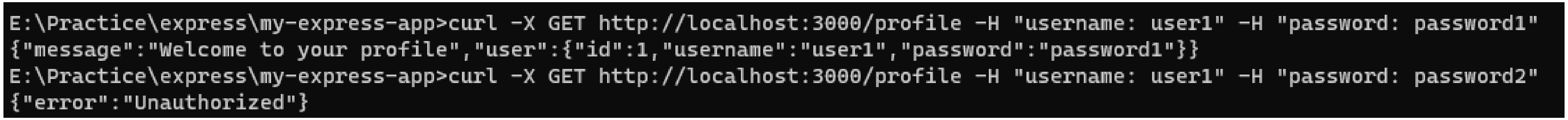
```
// Example of a protected route
app.get('/profile', authenticateUser, (req, res) => {
  // The user is authenticated and authorized to access this route
  res.json({ message: 'Welcome to your profile', user: req.user });
});

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

Example : Authentication using Middleware

- Test user authentication

curl -X GET http://localhost:3000/profile -H "username: user1" -H "password: password1"



```
E:\Practice\express\my-express-app>curl -X GET http://localhost:3000/profile -H "username: user1" -H "password: password1"
{"message":"Welcome to your profile","user":{"id":1,"username":"user1","password":"password1"}}
E:\Practice\express\my-express-app>curl -X GET http://localhost:3000/profile -H "username: user1" -H "password: password2"
{"error":"Unauthorized"}
```

- Note : in 2nd try user is user1, but password is password2. Thus it is unauthorised.

Example : Authentication using Middleware

- In this example
  - We define a middleware function called ***authenticateUser*** that checks user credentials provided in the request headers.
  - The ***authenticateUser*** middleware checks if the ***username*** and ***password*** provided in the headers match any user in the users array. If a match is found, the user is considered authenticated, and their user object is attached to the request object (***req***.***user***).
  - If authentication ***fails***, the middleware ***responds with*** a ***401*** Unauthorized status and an error message.
  - We create a protected route at /profile, which is only accessible to authenticated users. The ***authenticateUser*** middleware is added as a second argument to this route, ensuring that only authenticated users can access it.

Session

Sessions in Express.js

- Sessions in Express.js are a way to maintain state and user data across multiple HTTP requests.
- They enable web applications to remember user-specific information and provide a mechanism for user authentication, personalization, and data persistence.
- In Express.js, sessions are typically implemented using middleware such as "express-session."
- When a user visits a web application, a unique session is created for them, and a session ID is stored as a cookie on the user's browser.
- This session ID is used to associate subsequent requests from the same user with their session data stored on the server.

Sessions in Express.js

- Session middleware in Express manages the storage and retrieval of session data.
- You can use sessions to store user information, such as their username, user ID, or authentication status, and access that information throughout a user's interaction with the application.
- Sessions are crucial for implementing user authentication, as they allow the application to remember the user's identity and keep them logged in.

Setup and Installation

- To use sessions in Express, one must first install the express-session middleware and a session store like cookie-session or express-session for storing session data.

```
npm install express express-session
```

Setting Up Sessions

- In an Express application, sessions can be set up with the following code:

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

const app = express();

app.use(
  session({
    secret: 'mySecretKey',
    resave: false,
    saveUninitialized: true,
  })
);
```

Storing and Retrieving Session Data

- Once sessions are set up, the application can store and retrieve session data. For example, you can store the user's username in the session after they log in:

```
app.post('/login', (req, res) => {
  // Authenticate user...
  const username = 'exampleUser';

  // Store the username in the
session
  req.session.username = username;

  res.send('Logged in');
});
```

Storing and Retrieving Session Data

- Subsequently, you can access the stored username in other parts of the application, such as to personalize the user's experience:

```
app.get('/profile', (req, res) => {
  if (req.session.username) {
    const username = req.session.username;
    res.send(`Welcome, ${username}!`);
  } else {
    res.send('Not logged in');
  }
});
```

In this example, the session data (in this case, the username) is stored in the req.session object and can be accessed in other routes as needed.

Summary

- Sessions are a fundamental feature for building user-centric web applications.
- They enable applications to maintain user state, remember user actions, and provide a personalized experience. In a real-world application, sessions are often used in conjunction with user authentication mechanisms to create secure and dynamic web experiences.

End