SNS COLLEGE OF ENGINEERING

Kurumbapalayam(Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NAAC-UGC with 'A' Grade

Approved by AICTE, Recognized by UGC & Affiliated to Anna University, Chennai

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY

Course Code and Name : 19TS601 FULL STACK DEVELOPMENT

Unit 3 : NODEJS AND EXPRESS

Topic : EXPRESS –ROUTING-HANDLER FUNCTION

What is Express.js?

- Express.js is a streamlined web application framework for Node.js designed to facilitate the creation of web applications and APIs.
- It extends Node.js's core features, providing a structured approach to managing server-side logic.
- Known for its minimalistic approach, Express offers essential web application functionalities by default and enables developers to extend its features with middleware and plugins.

Features of Express.js

- Middleware
- Routing
- Template Engines
- Extensibility
- Performance

Features of Express.js

- **Middleware:** Middleware is essential in Express.
- It allows you to modify request and response objects, add processing logic, and handle errors effectively.
- **Routing:** Express offers a powerful routing mechanism to define application endpoints and handle various HTTP methods (GET, POST, PUT, DELETE, etc.).
- This feature simplifies the process of building RESTful APIs.

Features of Express.js

- **Template Engines**: Express supports various templating engines, such as Pug, EJS, and Handlebars.
- These engines enable you to generate dynamic HTML content on the server side.
- **Extensibility:** Express is highly extensible and can be integrated with numerous third-party libraries and tools.
- This flexibility allows you to easily add features like authentication, validation, and logging.
- **Performance:** Built on Node.js's asynchronous, non-blocking architecture, Express performs well when handling multiple simultaneous connections.

Why Use Express.js?

- **Simplicity**: Express abstracts the complexities of Node.js, helping developers build applications more quickly and with less code.
- **Flexibility:** Express's unopinionated nature allows you to structure your application as you see fit without imposing rigid conventions.
- **Community Support**: Express has a large and active community that provides numerous resources, tutorials, and third-party middleware to expand its capabilities.
- **Scalability**: Express is suitable for small-scale projects and large enterprise applications. It is versatile and can handle a wide range of use cases.

Routing

- Routing in Express.js defines URL patterns (endpoints) and links them with specific actions or resources within your web application.
- It allows users to navigate your application based on URLs. Express uses the app object to define routes.

Syntax:

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

Each route consists of three parts:

METHOD:

- Specifies the HTTP request method.
- Common methods include GET, POST, PUT, DELETE, etc.

PATH:

- Defines the URL pattern for the route. (e.g., /about, /contact).

HANDLER:

- The function that is executed when a route is matched.
- It typically sends a response back to the client.

Basic Routing with GET Requests

Basic routing with GET requests:

```
const express = require('express'); // Import express module
const app = express(); // Create express app
const port = 3000; // Define port

// Define a route for the root URL ('/') with a GET request method
app.get('/', (req, res) => {
    // Send a response to the client
    res.send('Welcome to the Home Page!');
});
```

```
// Define a route for the '/about' URL with a GET request method
app.get('/about', (req, res) => {
  // Send a response to the client
  res.send('About Us');
});
 // Define a route for the '/contact' URL with a GET request method
app.get('/contact', (req, res) => {
  // Send a response to the client
  res.send('Contact Us');
});
```

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

Explanation:

- The root route (/) handles GET requests for the root URL. When a request is received at this URL, the handler function sends 'Welcome to the Home Page!' as a response.
- The about route (/about) handles GET requests for the /about URL. When a GET request is made to /about, the handler function sends 'About Us' as a response.
- The contact route (/contact) handles GET requests for the /contact URL. When a request is received at this URL, the handler function sends 'Contact Us' as a response.

Route Parameters in Express.js

- In Express.js, route parameters are parts of the URL defined to capture dynamic values.
- These parameters are specified in the route path by prefixing a colon (:) before the parameter name.
- When a request is made to a route that includes parameters, Express.js extracts these values and makes them available in the req.params object.

Implementing Route Parameters in Express.js

- To define a route with parameters, use a colon (:) before the parameter name in the route path.

single route parameter: Example:

```
Copy Codeconst express = require('express');
const app = express();
// Define a route with a route parameter
app.get('/users/:userId', (req, res) => {
  // Access the route parameter value
  const userId = req.params.userId;
  res.send(`User ID: ${userId}`);
});
```

- // Start the server
- app.listen(3000, () => {
-     console.log('The server is running on port: 3000');
- });
- In the above example, when a user visits /users/1001, the application responds with "User ID: 1001". The req.params.userId retrieves the value from the URL and makes it available within the route handler.

Multiple Route Parameters

- Multiple parameters in a single route by separating them with slashes.

Example:

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

In the above example, visiting /users/1001/posts/2001 would respond with "User ID: 1001, Post ID: 2001".

Handling Optional Route Parameters

- There may be cases where a route parameter is not always required.
- Express.js supports optional route parameters by appending a question mark (?) to the parameter name.

Example:

```
app.get('/users/:userId/posts/:postId?', (req, res) => {
    const userId = req.params.userId;
    const postId = req.params.postId ? req.params.postId : 'No post ID
provided';
    res.send(`User ID: ${userId}, Post ID: ${postId}`);
});
```

In the above example, visiting /users/1001/posts will respond with "User ID: 1001, Post ID: No post ID provided", while visiting /users/1001/posts/2001 will respond with "User ID: 1001, Post ID: 2001".

Using Regular Expressions with Route Parameters

- Express.js also provides the facility to use regular expressions to define more flexible routes.
- This feature is particularly useful for validating the format of parameters. For Example:

```
app.get('/category/:categoryName([a-zA-Z]+)', (req, res) => {
  const categoryName = req.params.categoryName;
  res.send(`Category: ${categoryName}`);
});
```

- In the above example, the route /category/:categoryName ensures that the categoryName parameter contains only alphabetic characters.

- In Express.js, route handlers are a core feature that allows developers to define the behavior for specific HTTP requests.
- When building web applications, it's crucial to handle requests efficiently. Route handlers provide a way to map different request URLs and methods (such as GET, POST, PUT, and DELETE) to specific functions.

Route Handlers

- An Express.js route handler is a function that determines how the application responds to client requests at a particular endpoint.
- The endpoint is defined by a URL path and an HTTP method (GET, POST, PUT, DELETE, etc.).

- **Defining Routes for Different HTTP Methods**
- Express.js allows you to define route handlers for various HTTP methods. Below are examples of handling common HTTP requests like GET, POST, PUT, and DELETE.
- **Handling GET Requests**
- You use a GET request to retrieve data from the server. Here's an example of how to handle a GET request in Express.

Example:

```
// app.js
const express = require('express');
const app = express();
app.get('/', (req, res) => {
   res.send('Welcome to the homepage!');
});
app.get('/users', (req, res) => {
   // Simulate fetching user data
   const users = [{ id: 1, name: 'Alex' }, { id: 2, name: 'Jane' }];
   res.json(users);
```

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

- In this example, the root route (/) responds with a welcome message, and /users responds with a list of users in JSON format.

Handling POST Requests

- The POST method sends data to the server, typically when creating a new record. app.js

```
app.use(express.json()); // To parse JSON bodies
 app.post('/users', (req, res) => {
    const newUser = req.body;
    // Simulate adding a new user
    res.status(201).send(`New user ${newUser.name} added
successfully.`);
```

- In the above code, the server accepts a new user record sent in the request body and responds with a confirmation message.

Handling PUT Requests

- A PUT request to update existing data.
- The following example uses a route parameter (:id) to identify the user's record that needs updating.

Example:

- // app.js
- app.put('/users/:id', (req, res) => {
-     const userId = req.params.id;
-     const updatedData = req.body;
-     // Simulate updating user data
-     res.send(`User record with ID: ${userId} updated successfully.`);
- });
- In the above example, the route /users/:id allows the client to update a specific user's data based on the id parameter.

Handling DELETE Requests

- DELETE requests remove data from the server. Here's how you handle a DELETE request in Express.
- Example:app.js

```
app.delete('/users/:id', (req, res) => {
   const userId = req.params.id;


   // Simulate deleting user data
   res.send(`User record with ID: ${userId} deleted successfully.`);
});
```

In the above example, the server handles deleting a user's record identified by the id parameter.

Chaining Route Handlers

- In Express.js, you can chain route handlers for the same path but with different HTTP methods.
- This method eliminates redundancy when defining routes for similar paths. Using app.route(), you can group multiple methods (GET, POST, PUT) for a single endpoint, keeping the code clean and easy to manage.
- Example:

```
app.js
app.route('/user')
  .get((req, res) => {
    // Fetch user data
    res.send('Fetching user data');
  })
  .post((req, res) => {
    // Add a new user
    res.send('New user created');
  })
  .put((req, res) => {
    // Update user data
    res.send('User data updated');
  });
```

The above example /user route responds to GET, POST, and PUT requests in a streamlined manner.

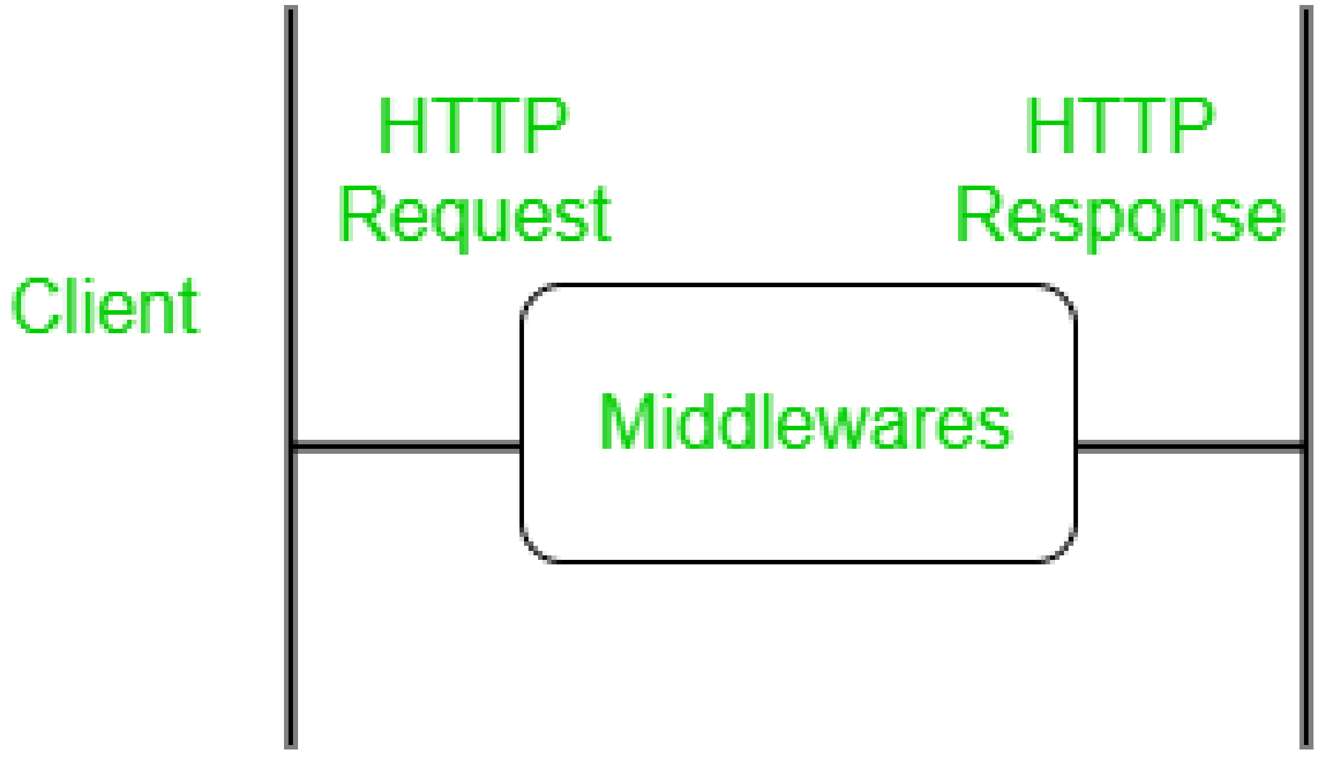
Middleware

- Middleware functions are the building blocks of any web server, especially in frameworks like ExpressJS.
- It plays a vital role in the request-response cycle.
- They are functions that have access to the request object (req), the response object (res), and the next function in the application’s request-response cycle.

What is Middleware in Express?

- Middleware in Express refers to functions that process requests before reaching the route handlers.
- These functions can modify request and response objects, end the request-response cycle, or call the next middleware function.

- Middleware functions are executed in the order they are defined.
- They can perform tasks like authentication, logging, or error handling.
- Middleware helps separate concerns and manage complex routes efficiently.


HTTP Request
HTTP Response
Client
Middlewares

Syntax

```
app.use((req, res, next) => {
    console.log('Middleware executed');
    next();
});
```

- (req, res, next) => {}: This is the middleware function where you can perform actions on the request and response objects before the final handler is executed.
- next(): This function is called to pass control to the next middleware in the stack if the current one doesn’t end the request-response cycle.

Middleware Works in Express.js

- In Express.js, middleware functions are executed sequentially in the order they are added to the application.
- When a request is received, it is passed through the middleware functions in the order they were defined.
- Each middleware can perform a task and either send a response or call the next() function to pass control to the next middleware function.

- Request arrives at the server.
- Middleware functions are applied to the request, one by one.
- Each middleware can either:
- Send a response and end the request-response cycle.
- Call next() to pass control to the next middleware.
- If no middleware ends the cycle, the route handler is reached, and a final response is sent.

Types of Middleware

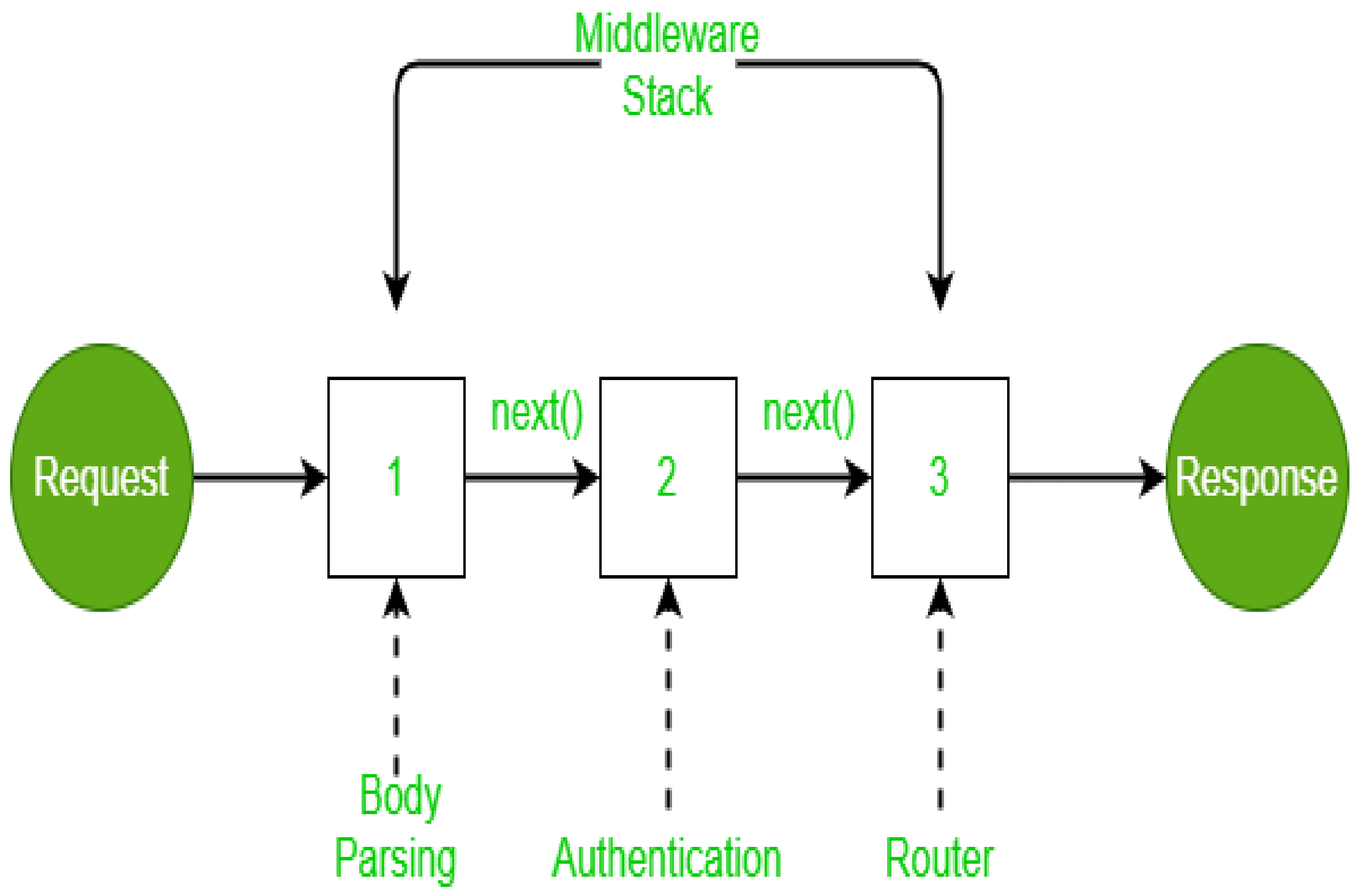
- ExpressJS offers different types of middleware :
- **Application-level middleware:** Bound to the entire application using app.use() or app.METHOD() and executes for all routes.
- **Router-level middleware:** Associated with specific routes using router.use() or router.METHOD() and executes for routes defined within that router.
- **Error-handling middleware:** Handles errors during the request-response cycle. Defined with four parameters (err, req, res, next).
- **Built-in middleware:** Provided by Express (e.g., express.static, ExpressJSon, etc.).
- **Third-party middleware:** Developed by external packages (e.g., body-parser, morgan, etc.).

Advantages of using Middleware

- **Modularity:** Breaks down complex tasks into smaller, manageable functions.
- **Reusability:** Middleware functions can be reused across different routes or applications.
- **Maintainability:** Organizes code logically, making it easier to manage and update.
- **Error Handling:** Centralizes error handling, improving the application’s robustness.
- **Performance Optimization:** Allows for tasks like caching, compression, and security checks to be handled efficiently.

Middleware Chaining

- Middleware can be chained from one to another, Hence creating a chain of functions that are executed in order. The last function sends the response back to the browser. So, before sending the response back to the browser the different middleware processes the request.
- The next() function in the express is responsible for calling the next middleware function if there is one.
- **Modified requests will be available to each middleware via the next function**
-


Middleware Stack
Request
1
next()
2
next()
3
Response
Body Parsing
Authentication
Router

```
const express = require('express');
const app = express();
// Middleware 1: Log request method and URL
app.use((req, res, next) => {
   console.log(`${req.method} request to ${req.url}`);
   next();
});
// Middleware 2: Add a custom header
app.use((req, res, next) => {
   res.setHeader('X-Custom-Header', 'Middleware Chaining Example');
   next();
});
```

```
// Route handler
app.get('/', (req, res) => {
  res.send('Hello, World!');
});
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Middleware 1: Logs the HTTP method and URL of the incoming request.

Middleware 2: Sets a custom header X-Custom-Header in the response.

Route Handler: Sends a “Hello, World!” message as the response.

Output

When a client makes a GET request to http://localhost:3000/, the server responds with:

```
Hello, World!
```

Thank
You!