Express part 02 (Express middleware)

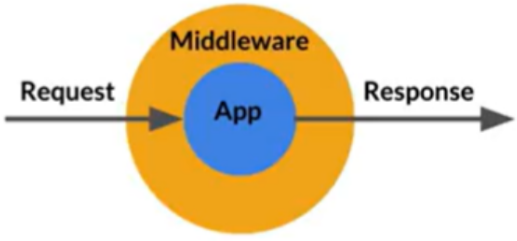
Lesson Plan

PW SKILLS

Topics

- What is a middleware?
- Need for using middleware
- Types of middleware
- Middleware chaining
- Where can we apply the middleware
- Ordering of the middleware

What is middleware?



Middleware is a crucial concept that allows you to enhance and modify the request and response objects in the HTTP cycle. Middleware functions 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.

For example

```
Unset
const myMiddleware = (req, res, next) => {
  // Do something with req and res
  // Call the next middleware in the stack
  next();
};
```

- **req (request):** This object represents the HTTP request made by a client (e.g., a web browser) to the server. It contains information about the client, the request method, headers, and any data sent with the request (such as form data or JSON payloads).
- **res (response):** This object represents the HTTP response that the server sends back to the client. It allows you to set headers, status codes, and send data back to the client, such as HTML content, JSON, or other types of responses.
- **next:** This is a function that, when called, passes control to the next middleware function in the stack. Middleware functions can modify the req and res objects or terminate the request-response cycle by sending a response to the client. Calling next() is crucial to moving the request through the middleware chain.

Need for using middleware

- It can be used to process request before they are processed inside the application
- It can be used to add additional functionalities such as logging and authentication it can be used to apply some filters on the requests
- It can be used to set some headers to both requests and response

Types of middleware

In Express.js, middleware functions are categorized based on their scope and purpose. Here are some common types of middleware in Express:

1. **Application-Level Middleware:**
   app.use() middleware: Applied to the entire application. Executes for every incoming request. Used for global configurations, setting headers, or handling tasks common to all routes.

```
Unset
app.use((req, res, next) => {
  // Middleware logic
  next();
});
```

2. **Route-Level Middleware:**
   - Middleware for a specific route or route group: Applied to a specific route or a group of related routes. Can be associated with one or more HTTP methods.

```
Unset
app.get('/route', (req, res, next) => {
  // Middleware specific to the /route
  next();
});
```

3. **Error-Handling Middleware:**
   - Middleware to handle errors: Special middleware with four parameters (err, req, res, next). Used to catch and handle errors that occur during the request-response cycle.

```
Unset
app.use((err, req, res, next) => {
  // Error handling logic
  res.status(500).send('Internal Server Error');
})
```

4. Built-in Middleware:

- Express provides several built-in middleware functions: Such as express.static for serving static files, express.json for parsing JSON in requests, and express.urlencoded for parsing URL-encoded data.

```
Unset
app.use(express.static('public'));
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

5. Third-Party Middleware:

- Middleware provided by external packages: Examples include body-parser for parsing request bodies and morgan for request logging.

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

app.use(bodyParser.json());
app.use(morgan('combined'));
```

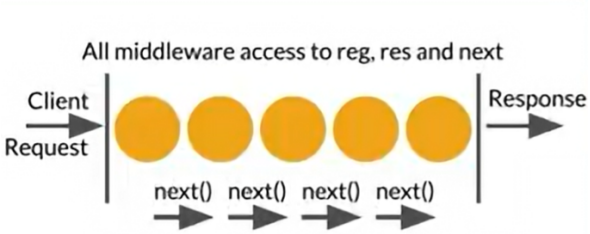
6. Custom Middleware:

- User-defined middleware functions: Created to fulfill specific requirements. Custom middleware can be applied at different levels in the application.

```
Unset
const customMiddleware = (req, res, next) => {
  // Custom middleware logic
  next();
};

app.use(customMiddleware);
```

Middleware chaining



Middleware chaining typically refers to the process of combining multiple middleware functions in a specific order to handle an incoming HTTP request. Middleware functions are executed in the order they are defined in the code, and each middleware function can modify the request or response before passing control to the next middleware in the chain.

Example

```
Unset
var express = require("express");
var app = express();

app.use(function(req, res, next) {
  console.log("Request received at " + Date.now());
  next();
});

// Start the server
app.listen(8000, function() {
  console.log("Server is running on port 3000");
});
```

Middleware 1: Logs "Middleware 1" to the console and passes control to the next middleware.

Middleware 2: Logs "Middleware 2" to the console and passes control to the next middleware.

Route handler: Responds with "Hello, world!" when a GET request is made to the root path ('/').

When you run this code and make a request to the server, you'll see the middleware functions being executed in the order they are defined, followed by the route handler. The console output might look like this:

```
Unset
Middleware 1
Middleware 2
Server is running on port 3000
```

Where can we apply the middleware

Middleware can be applied at the application level and the route level. This flexibility allows you to control the flow of requests globally for the entire application or selectively for specific routes. Let's explore both types:

Application-Level Middleware:

Application-level middleware is applied to the entire Express application. It gets executed for every incoming request, regardless of the specific route. You can use app.use() to apply middleware at the application level. This is useful for tasks that need to be performed globally, such as setting up common configurations, handling authentication, or logging.

Example

```
Unset
var express = require("express");
var app = express();

app.use(function(req, res, next) {
  console.log("Request received at " + Date.now());
  next();
});

// Start the server
app.listen(8000, function() {
  console.log("Server is running on port 3000");
});
```

Route-Level Middleware:

Route-level middleware is applied to specific routes or groups of routes. You can use app.METHOD() (e.g., app.get(), app.post()) to apply middleware to a particular route or set of routes. Alternatively, you can use app.use() without specifying a method to define route-level middleware. This flexibility allows you to customize the behavior for specific paths, methods, or groups of related routes.

Example

```
Unset
var express = require("express");
var app = express();

app.use("/students", function(req, res, next) {
  console.log("Request received at " + Date.now());
  next(); // Don't forget to call next to pass control to the
next middleware or route handler
});

// Start the server
app.listen(3000, function() {
  console.log("Server is running on port 3000");
});
```

Ordering of the middleware

The order of middleware in Express.js matters, as middleware functions are executed sequentially in the order they are defined. This order allows you to control the flow of the request-response cycle. Here's an example to illustrate the importance of the order:

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

// Middleware 1
app.use((req, res, next) => {
  console.log('Middleware 1 - Before Route Handler');
  next();
});

// Route handler
app.get('/', (req, res, next) => {
  console.log('Route Handler');
  next();
});

// Middleware 2
app.use((req, res, next) => {
  console.log('Middleware 2 - After Route Handler');
  next();
});

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

Middleware 1: Executed before the route handler. Logs "Middleware 1 - Before Route Handler."

Route handler: Logs "Route Handler" and then calls the next middleware.

Middleware 2: Executed after the route handler. Logs "Middleware 2 - After Route Handler."

When you run this code and make a request to the server, you'll see the middleware functions being executed in the order they are defined, followed by the route handler. The console output might look like this:

```
Unset
Middleware 1 - Before Route Handler
Route Handler
Middleware 2 - After Route Handler
Server is running on port 3000
```

The order of middleware in Express is essential for controlling the request and response flow. Middleware before the route handler is executed on the way in, and middleware after the route handler is executed on the way out. Adding more functions or route handlers allows for sequential execution, providing a powerful way to customize and structure request handling in your Express.js application