Middleware Basics,

https://youtu.be/_qfozHD3x4I

Middleware Basics

Introduction to Middleware

What is Middleware?

Definition:

Middleware functions are those that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's requestresponse cycle.

These functions can:

Execute any code.

Make changes to the request and response objects.

End the requestresponse cycle.

Call the next middleware function in the stack.

Context in MERN Stack:

In a MERN stack application, middleware is primarily used in the serverside part with Express.js.

Middleware acts as a bridge between incoming requests and the corresponding responses, ensuring smooth and controlled data flow.

Types of Middleware

1. Applicationlevel Middleware:

Bound to an instance of the Express application using `app.use()` or `app.METHOD()`, where `METHOD` is an HTTP method.

Useful for tasks such as logging, authentication, and handling common functionality across different routes.

2. Routerlevel Middleware:

Similar to applicationlevel middleware but bound to an instance of `express.Router()`.

Helps in organizing and modularizing the middleware functionality within different routes.

3. Errorhandling Middleware:

Defined with four arguments (err, req, res, next).

Specifically designed to catch and handle errors that occur during the requestresponse cycle.

4. Builtin Middleware:

Express.js provides some builtin middleware functions like `express.json()` and `express.urlencoded()` for common tasks.

express.json(): Parses incoming requests with JSON payloads.

express.urlencoded(): Parses incoming requests with URLencoded payloads.

5. Thirdparty Middleware:

Numerous thirdparty middleware packages are available to integrate into Express applications for added functionality.

Examples include:

CORS: Handling CrossOrigin Resource Sharing.

Session: Managing user sessions.

Cookies: Parsing and handling cookies.

Why Use Middleware?

Code Organization:

Middleware helps separate different concerns like authentication, logging, and error handling into distinct modules.

This separation enhances the readability and maintainability of the code.

Reusability:

Middleware functions can be reused across different routes and applications, promoting DRY (Don't Repeat Yourself) principles.

Maintainability:

Updating specific functionalities becomes easier as middleware allows you to change one part of the code without affecting the rest of the application.

Scalability:

Middleware enables the efficient handling of common tasks across various parts of an application, which aids in scaling the application.

Example Usage

Basic Example:

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

const app = express();


// Middleware function to log request details

function logRequest(req, res, next) {

   console.log(`${req.method} ${req.url}`);

   next(); // Pass control to the next middleware
}

// Applicationlevel middleware
app.use(logRequest);

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

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Explanation:

`logRequest` is a simple middleware function that logs the HTTP method and URL of each incoming request.

By using `app.use(logRequest)`, this middleware is applied to all routes in the application.

The `next()` function passes control to the next middleware function in the stack, allowing for multiple middleware functions to handle different aspects of a request.

Conclusion:

Middleware is a cornerstone of Express.js applications and, by extension, MERN stack development.

It empowers developers to write clean, modular, and efficient code.

By mastering middleware, developers can significantly enhance the performance and maintainability of their web applications.

ExpressRoute method -GET,
https://youtu.be/Jr21ErSwUBc

Introduction to GET Method

HTTP GET Method:

The GET method is one of the most common HTTP methods.

It is used to request data from a specified resource.

GET requests can retrieve data, but they should not change any data on the server.

Setting Up an Express Application

1. Install Node.js and Express:

Ensure Node.js is installed. Download from nodejs.org if not already installed.

Create a new directory for your project.

```
mkdir expressgetdemo
cd expressget
```

What is GET?

In web development, GET is one of the most commonly used HTTP methods. It is used to request data from a specified resource. When you type a URL into your web browser and press enter, your browser sends a GET request to the server to retrieve the content of that webpage.

Express.js and the GET Method

Express.js, a popular web application framework for Node.js, makes handling GET requests straightforward and efficient. In Express, the app.get() method is used to define a route that listens for GET requests.

How Does it Work?

When you define a GET route, you specify a path and a callback function. This callback function takes three arguments: req, res, and next. Here's a simple example:

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

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

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
   console.log(Server is running on port ${PORT});
});
```

Explanation of the Code

- const express = require('express'); - This line imports the Express module.

- const app = express(); - We create an instance of an Express application.

- app.get('/', (req, res) => { res.send('Hello, World!'); }); - Here, we define a GET route for the root URL (/). When this URL is accessed, the server responds with 'Hello, World!'.

- const PORT = process.env.PORT || 3000; - This sets the port the server will listen on.

- app.listen(PORT, () => { console.log(Server is running on port ${PORT}); }); - This starts the server and listens for connections on the specified port.

Why Use GET Routes?

GET routes are essential for retrieving data and displaying web pages. They are widely used for:

- Fetching data from a database.

- Displaying information to users.

- Navigating through different pages of a website.

Conclusion

Understanding the GET method in Express.js is crucial for any web developer. It allows you to handle requests efficiently and provide users with the data they need. I encourage you to experiment with defining your own GET routes to see the power and simplicity of Express.js in action.

ExpressRoute method -POST,

https://youtu.be/nlhLP8V-w_4

Introduction to POST Method

HTTP POST Method:

The POST method is used to send data to the server to create a new resource.

Unlike GET requests, POST requests can include a body, allowing you to send data such as form submissions.

Setting Up an Express Application

Install Node.js and Express:

Ensure Node.js is installed. Download from nodejs.org if not already installed.

Create a new directory for your project.

```
mkdir express-post-demo
cd express-post-demo
```

Initialize a new Node.js project.

```
npm init -y
```

Install Express.

```
npm install express
```

Create the Application File:

Create a file named app.js and set up a basic Express application.

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

// Middleware to parse JSON bodies
app.use(express.json());

// POST route
app.post('/submit-form', (req, res) => {
  const formData = req.body;
  // Process the form data here
  res.send('Form submitted successfully!');
});

// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

How to Test the POST Method

Using Postman or cURL:

Postman:

Open Postman and create a new POST request.

Set the URL to http://localhost:3000/submit-form.

In the Body tab, select raw and JSON format, then add JSON data.

Example:

```
{
  "name": "John Doe",
  "email": "john.doe@example.com"
}
```

Send the request and observe the response.

ExpressRoute method -PUT,

https://youtu.be/3TrzM8N0u1U

Introduction to PUT Method

HTTP PUT Method:

The PUT method is used to update an existing resource on the server.

It is idempotent, meaning that multiple identical requests will have the same effect as a single request.

Setting Up an Express Application

Install Node.js and Express:

Follow the same steps as above to set up a new Express application.

Create the Application File:

Create a file named app.js and set up a basic Express application.

javascript

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

// Middleware to parse JSON bodies
app.use(express.json());

// PUT route
app.put('/update-user/:id', (req, res) => {
   const userId = req.params.id;
   const updatedData = req.body;
   // Update user data here
   res.send(`User ${userId} updated successfully!`);
});

// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
   console.log(`Server is running on port ${PORT}`);
});
```

How to Test the PUT Method

Using Postman or cURL:

Postman:

Open Postman and create a new PUT request.

Set the URL to http://localhost:3000/update-user/1.

In the Body tab, select raw and JSON format, then add JSON data.

Example:

```json
{
  "name": "Jane Doe",
  "email": "jane.doe@example.com"
}
```

Send the request and observe the response.

Conclusion

The PUT method is crucial for updating existing resources on the server.

It ensures that the specified resource is updated with the provided data, making it ideal for APIs that require modifications to existing data.

ExpressRoute method -DELETE,

https://youtu.be/fyItqQuzCzM*

Introduction to DELETE Method

HTTP DELETE Method:

The DELETE method is used to delete a resource from the server.

It is used to remove items from a database or files from a server.

Setting Up an Express Application

Install Node.js and Express:

Follow the same steps as above to set up a new Express application.

Create the Application File:

Create a file named app.js and set up a basic Express application.

javascript

Copy code

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

// DELETE route
app.delete('/delete-user/:id', (req, res) => {
  const userId = req.params.id;
  // Delete user data here
  res.send(`User ${userId} deleted successfully!`);
});

// Start the server
```

```
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

How to Test the DELETE Method

Using Postman or cURL:

Postman:

Open Postman and create a new DELETE request.

Set the URL to http://localhost:3000/delete-user/1.

Send the request and observe the response.

Conclusion

The DELETE method is essential for removing resources from the server.

It allows for the deletion of specified resources, helping to maintain clean and updated data on the server.

Route paths-Strings ,

https://youtu.be/7fArMnGvY60

Title: Understanding Route Paths in Express.js

Introduction:

Section 1: What are Route Paths?

Definition:

"Route paths define endpoints at which requests can be made in your web application. In Express.js, routes are used to determine how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, PUT, DELETE, etc.)."

Types of HTTP Methods:

"In Express.js, the following HTTP methods are commonly used with route paths:

GET: Retrieve data from the server.

POST: Submit data to be processed to the server.

PUT: Update existing data on the server.

DELETE: Delete data from the server.

Each of these methods corresponds to a different operation you might want to perform on a resource."

Why Route Paths Matter:

"Route paths are crucial because they define the structure of your web application's URL scheme. This structure helps you organize your code better and makes your application more intuitive for users."

Section 2: String Route Paths

Introduction to String Route Paths:

"In Express.js, string route paths are the most straightforward way to define routes. They are literal strings that specify the exact path the server will respond to."

Examples of Simple Routes:

"Let's look at a few examples:

Root route: '/' This represents the home page or the root of your web application.

Specific route: '/about' This could be an about page where users can learn more about your application or organization.

Nested routes: '/user/profile' This might represent a user's profile page within the application."

Benefits of Using String Route Paths:

"Using string route paths makes your code easy to read and maintain. It's straightforward and there's no ambiguity about what path a particular route refers to."

Section 3: Creating Routes Using Strings

Setting up the Project:

"Before we dive into coding, let's ensure we have a basic Express.js project set up. If you haven't set up your project yet, make sure you have Node.js installed and then create a new directory for your project. Initialize it with npm init and install Express using npm install express."

Installing Express:

```
npm install express
```

Creating the Server:

File Structure:

"Create a file named server.js. This will be our main server file where we'll define our routes."

Code Example:

javascript

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

// Root route
app.get('/', (req, res) => {
    res.send('Welcome to the Home Page!');
});

// About route
app.get('/about', (req, res) => {
    res.send('This is the About Page.');
});

// User profile route
app.get('/user/profile', (req, res) => {
    res.send('Welcome to your Profile!');
});
```

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

Explanation of Code:

"In this example, we are creating three routes using string paths:

Root route: app.get('/', (req, res) => { ... }); This responds to a GET request at the root URL / with a message 'Welcome to the Home Page!'.

About route: app.get('/about', (req, res) => { ... }); This responds to a GET request at the URL /about with a message 'This is the About Page.'.

User profile route: app.get('/user/profile', (req, res) => { ... }); This responds to a GET request at the URL /user/profile with a message 'Welcome to your Profile!'."

Section 4: Testing the Routes

Starting the Server:

```
node server.js
```

Accessing the Routes:

"Open your web browser and navigate to the following URLs to test the routes:

http://localhost:3000/ You should see the message 'Welcome to the Home Page!'.

http://localhost:3000/about You should see the message 'This is the About Page.'.

http://localhost:3000/user/profile You should see the message 'Welcome to your Profile!'."

Troubleshooting Tips:

"If you encounter any issues, ensure that your server is running and that you have typed the URLs correctly. Also, check the terminal for any error messages that might indicate what went wrong."

Route paths- Patterns & Regular Expressions,
https://youtu.be/RYRdNRkMnH4

String Patterns & Regular Expressions

Wildcards: * can be used to match any number of characters, making routes more flexible.

Route Parameters: :param can be used to capture part of the URL and make it available in req.params. This is useful for dynamic routes like user profiles, product pages, etc.

Optional Parameters: Parameters can be optional by adding a ? after the parameter name (e.g., /:id?).

Regular Expressions

Complex Patterns: Regular expressions allow for defining complex and precise patterns for route matching.

Use Cases: Useful for validation and strict matching requirements, such as only allowing numeric values, specific string patterns, etc.

Performance: Regular expression matching can be slower than simple string matching, so use them judiciously.

Example Code

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

const app = express();


// Exact match

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

  res.send('About us');

});


// String pattern with wildcard

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

  res.send('Blog article');
});

// String pattern with route parameter
app.get('/user/:username', (req, res) => {
  res.send(`User profile of ${req.params.username}`);
});

// Regular expression match
app.get(/^\/product\/\d+$/, (req, res) => {
  res.send('Product with numeric ID');
});

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

In this example:

/about matches exactly.

/blog/* matches any path starting with /blog/.

/user/:username captures the username parameter.

/product/\d+ matches paths like /product/123 where the ID is numeric.

Summary

Understanding and utilizing different types of route paths in Express.js is essential for building dynamic and efficient web applications. String paths are simple and straightforward, string patterns offer flexibility with wildcards and parameters, and regular expressions provide the ultimate control over route matching. Use these tools appropriately to create robust and maintainable routes in your applications.

Route Parameters,

https://youtu.be/DSiOH46lMv4

Express.js Route Parameters

Introduction to Route Parameters

Route parameters in Express.js allow you to capture values specified at a particular position in the URL. They are used to identify a resource within the API.

Basic Usage of Route Parameters

Syntax: Route parameters are defined using a colon : followed by the parameter name.

Example:

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

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

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

Explanation:

:userId is the route parameter.

req.params is an object containing the values of the named parameters.

Access the parameter value using req.params.userId

Multiple Route Parameters

Syntax: Multiple parameters can be used in a single route.

Example:

```
 app.get('/user/:userId/book/:bookId', (req, res) => {
    const { userId, bookId } = req.params;
    res.send(User ID is ${userId} and Book ID is ${bookId});
 });
```

Explanation:

 Parameters :userId and :bookId are captured.

 Destructure req.params to access multiple parameters.

Optional Parameters

Syntax: Adding a question mark ? after the parameter name makes it optional.

Example:

```
 app.get('/user/:userId/book/:bookId?', (req, res) => {
    const { userId, bookId } = req.params;
    if (bookId) {
       res.send(User ID is ${userId} and Book ID is ${bookId});
    } else {
       res.send(User ID is ${userId}, no Book ID provided);
    }});
```

Explanation:

:bookId? is optional.

The route /user/123/book/ will match and bookId will be undefined.

Regular Expression Route Parameters

Syntax: Regular expressions can be used to define constraints on the route parameters.

Example:

```
app.get('/user/:userId(\\d+)', (req, res) => {
    const userId = req.params.userId;
    res.send(User ID is ${userId});
});
```

Explanation:

The route parameter :userId(\\d+) ensures that userId consists of digits only.

Routes like /user/123 will match, but /user/abc will not.

Handling Route Parameters in Middleware

Middleware: You can use middleware to handle route parameters before passing control to the next route handler.

Example:

```
app.param('userId', (req, res, next, userId) => {
    // Perform operations with userId
    req.user = { id: userId, name: 'John Doe' }; // Example
```

```
  next();
});

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

Explanation:

app.param is used to define middleware for userId.

Middleware function processes userId and adds user info to req.

Nested Routes and Route Parameters

Nested Routes: Express allows you to create nested routes, and route parameters can be used within these routes.

Example:

```
const userRouter = express.Router();

userRouter.get('/:userId', (req, res) => {
  const { userId } = req.params;
  res.send(User ID is ${userId});
});

userRouter.get('/:userId/book/:bookId', (req, res) => {
  const { userId, bookId } = req.params;
  res.send(User ID is ${userId} and Book ID is ${bookId});
});
```

```
app.use('/user', userRouter);
```

Explanation:

userRouter handles nested routes for /user.

Route parameters :userId and :bookId are used within userRouter.

Dynamic Route Parameters with Regex and Named Parameters

Regex and Named Parameters: Combine regex patterns with named parameters for more flexibility.

Example:

```
app.get('/user/:userId(\\d+)', (req, res) => {
    const userId = req.params.userId;
    res.send(User ID is ${userId});
});

app.get('/user/:userId/book/:bookId([azAZ]+)', (req, res) => {
    const { userId, bookId } = req.params;
    res.send(User ID is ${userId} and Book ID is ${bookId});
});
```

Explanation:

:userId(\\d+) captures numeric userId.

:bookId([azAZ]+) captures alphabetic bookId.

Best Practices for Using Route Parameters

1. Consistency: Use consistent naming conventions for parameters.

2. Validation: Validate and sanitize parameter values to prevent security issues.

3. Documentation: Clearly document the expected route parameters for each route.

4. Error Handling: Implement error handling for invalid or missing parameters.

Route Handlers: Single Function ,

https://youtu.be/cAzR_wgLfbQ

Route Handlers: Single Function in Express.js

Route handlers in Express.js are functions that manage the response to incoming requests. For simple operations, you can use a single function to handle these routes.

Setting Up a Basic Express Server

1. Initialize your Node.js project:

```
bash
mkdir expressapp
cd expressapp
npm init y
```

2. Install Express.js:

```
bash
npm install express
```

3. Create a basic Express server:

```
javascript
// app.js
const express = require('express');
const app = express();

// Define a route handler for the root URL (/)
```

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

// Define a port for the server to listen on
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
   console.log(Server is running on port ${PORT});
});
```

Explanation

1. Setup:

Create a directory for your project and initialize a Node.js project using npm init y.

Install Express.js using npm install express.

2. Server Creation:

Import Express and create an instance of it (const app = express();).

Define a route handler for the root URL (/). This route uses a single function to send a response "Hello, Express!".

Define a port for the server to listen on (const PORT = process.env.PORT || 3000;).

Start the server and log a message indicating that the server is running.

Additional Examples

1. GET Request to /greet:

   javascript

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

2. POST Request to /submit:

   javascript

```
// To handle POST requests, you'll need bodyparser middleware
app.use(express.json());

app.post('/submit', (req, res) => {
    const data = req.body;
    res.send(Data received: ${JSON.stringify(data)});
});
```

3. Dynamic Route for User Profiles:

   javascript

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

Testing the Routes

1. Run the server:

```bash
node app.js
```

2. Test the endpoints:

   Open a browser and go to http://localhost:3000 to see "Hello, Express!".

   Go to http://localhost:3000/greet to see "Greetings from Express!".

   Use a tool like Postman to send a POST request to http://localhost:3000/submit with a JSON body to see the data received.

   Go to http://localhost:3000/user/123 to see "User Profile for User ID: 123".

Conclusion

Using single function route handlers in Express.js is straightforward and effective for simple routes. This approach keeps your code clean and easy to understand, especially for basic operations.

Route Handlers: Array of Functions,

https://youtu.be/8ggPg0uqytk

Route Handlers: Array of Functions in Express.js

In Express.js, you can handle routes with an array of functions, allowing you to modularize your middleware and keep your code clean and manageable. This technique is particularly useful for complex routes that require multiple steps of processing.

Basic Example

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

// Middleware functions
const middleware1 = (req, res, next) => {
  console.log('Middleware 1');
  next();
};

const middleware2 = (req, res, next) => {
  console.log('Middleware 2');
  next();
};

const handler = (req, res) => {
  res.send('Hello from the handler!');
};

// Route using an array of functions
```

```
app.get('/example', [middleware1, middleware2, handler]);

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

In this example, the `/example` route passes through `middleware1`, `middleware2`, and then the final `handler`.

Practical Use Case: Authentication and Logging

A common use case for an array of middleware functions is to perform authentication and logging before handling the main request.

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

// Authentication middleware
const authenticate = (req, res, next) => {
  const token = req.headers['authorization'];
  if (token === 'secret-token') {
    next();
  } else {
    res.status(401).send('Unauthorized');
  }
};

// Logging middleware
const logRequest = (req, res, next) => {
  console.log(`Request to ${req.url} at ${new Date().toISOString()}`);
  next();
};

// Request handler
const getUserProfile = (req, res) => {
  res.json({ user: 'John Doe', age: 30 });
};

// Route using an array of middleware functions
app.get('/profile', [authenticate, logRequest, getUserProfile]);

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

In this case, the `/profile` route first checks if the user is authenticated with the `authenticate` middleware. If authenticated, it logs the request using `logRequest` middleware before finally sending the user profile with the `getUserProfile` handler.

Error Handling Middleware

You can also include error handling middleware in the array. Error handling middleware must have four arguments: `(err, req, res, next)`.

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

// Middleware functions
const checkQuery = (req, res, next) => {
  if (req.query.valid) {
    next();
  } else {
    next(new Error('Invalid query parameter'));
  }
};

const logError = (err, req, res, next) => {
  console.error(err.message);
  next(err);
};

const errorHandler = (err, req, res, next) => {
  res.status(400).send(err.message);
};

// Route using an array of functions including error handling
app.get('/error-handling', [checkQuery, logError, errorHandler]);
```

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

In this example, if the query parameter `valid` is not present, the `checkQuery` middleware throws an error, which is then logged by `logError` and finally handled by `errorHandler`.

Summary

Using an array of functions for route handling in Express.js provides a flexible and powerful way to manage complex request-processing logic. It allows you to separate concerns, improve code readability, and maintainability. This technique is ideal for scenarios requiring multiple processing steps like authentication, logging, validation, and error handling.

Route Handlers: Combination of Function and Array of Functions ,
https://youtu.be/V9CXDsRLuUs

Route Handlers: Combination of Function and Array of Functions in Express.js

In Express.js, route handlers are functions that handle requests to specific routes (URLs) and define how to respond to those requests. Express allows you to use a combination of single functions and arrays of functions as route handlers. This flexibility lets you create modular and reusable middleware for handling requests.

Basic Route Handler

A simple route handler is a single function that takes req, res, and next as parameters:

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

Using Multiple Functions

Express allows you to define multiple middleware functions for a single route. These functions are executed in sequence.

```
app.get(
  '/example',
  (req, res, next) => {
    console.log('First middleware');
    next();
  },
  (req, res) => {
    res.send('Hello, World!');
  }
);
```

Using an Array of Functions

You can also pass an array of middleware functions. This is useful for reusing common middleware across different routes.

```
const logRequest = (req, res, next) => {
  console.log(${req.method} ${req.url});
  next();
};

const sendResponse = (req, res) => {
  res.send('Hello, World!');
};

app.get('/example', [logRequest, sendResponse]);
```

Combining Functions and Arrays

You can combine single functions and arrays of functions in your route handlers for more complex request processing.

```
const validateUser = (req, res, next) => {
  if (req.query.user === 'admin') {
    next();
  } else {
    res.status(403).send('Forbidden');
  }
};

const logRequest = (req, res, next) => {
  console.log(${req.method} ${req.url});
  next();
};

const sendResponse = (req, res) => {
  res.send('Hello, World!');
};

app.get('/example', validateUser, [logRequest, sendResponse]);
```

Practical Example: User Authentication

Let's create a practical example where we combine middleware functions to handle user authentication and logging.

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

const authenticateUser = (req, res, next) => {
  if (req.headers.authorization === 'Bearer mysecrettoken') {
    next();
  } else {
    res.status(401).send('Unauthorized');
  }
};

const logRequest = (req, res, next) => {
  console.log(${req.method} ${req.url});
  next();
};

const sendResponse = (req, res) => {
  res.send('Welcome, authenticated user!');
};

app.get('/secure', authenticateUser, [logRequest, sendResponse]);

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

Summary

Single Function: A straightforward way to handle a route with one function.

Multiple Functions: Allows for sequential execution of multiple middleware functions.

Array of Functions: Useful for modular and reusable middleware.

Combining: Mix single functions and arrays for complex request handling.

Response Methods : download() and end(),

https://youtu.be/YU5yIkipqcA

In Express.js, the response object provides several methods to handle the HTTP response that will be sent back to the client. Two such methods are download() and end(). Here's an overview of these methods:

response.download()

The download() method is used to prompt the client to download a file. It sets the appropriate headers to force the browser to download the file rather than display it. This method is particularly useful when you want to allow users to download files directly from your server.

Syntax:

javascript

```
res.download(path [, filename] [, options] [, callback])
```

path: The path to the file you want to download.

filename (optional): The name to be used for the downloaded file. If not provided, the default is the filename from path.

options (optional): An object that can include headers and other options.

callback (optional): A callback function that will be called after the download starts.

Example:

javascript

```
app.get('/download', (req, res) => {
  const filePath = 'path/to/file.txt';
  res.download(filePath, 'customfilename.txt', (err) => {
    if (err) {
      console.error('Error during file download:', err);
      res.status(500).send('An error occurred while downloading the file.');
    }
  });
});
```

response.end()

The end() method is used to end the response process. It signals to the server that all of the response headers and body have been sent, and that the server should consider the request complete. This method is typically used when you have finished sending data through the response object.

Syntax:

javascript

```
res.end([data] [, encoding] [, callback])
```

data (optional): The data to be sent as the body of the response.

encoding (optional): The encoding of the data.

callback (optional): A callback function that will be called when the response is finished.

Example:

javascript

```
app.get('/end', (req, res) => {
  res.write('Hello, World!');
  res.end();
});
```

Using end() with data:

javascript

```
app.get('/endwithdata', (req, res) => {
  res.end('Goodbye, World!');
});
```

Summary

Use res.download() to facilitate file downloads for clients, allowing users to download files from your server.

Use res.end() to finish the response process, signaling that all headers and body data have been sent.

These methods provide flexibility in handling different types of responses in your Express.js applications.

Response Methods : json() and redirect(), https://youtu.be/ixFd46JDFGc

In Express.js, the json() and redirect() methods are used to handle responses in different ways. Here's a detailed look at both:

json()

The json() method is used to send a JSON response. It sets the ContentType header to application/json and sends a JSONformatted response body. This method is useful when you need to send structured data to the client, such as API responses.

Syntax

```
res.json([body])
```

Example

```
app.get('/user', (req, res) => {
  const user = {
    id: 1,
    name: 'John Doe',
    email: 'john.doe@example.com'
  };
  res.json(user);
});
```

In this example, when a GET request is made to /user, the server responds with a JSON object containing the user's details.

redirect()

The redirect() method is used to redirect the client to a different URL. This method sets the status code to 302 (Found) by default, which indicates a temporary redirect, but you can also specify other status codes like 301 (Moved Permanently).

Syntax

```
res.redirect([status], path)
```

status (optional): The HTTP status code for the redirect.

path: The URL to which the client should be redirected.

Example

```
app.get('/oldroute', (req, res) => {
  res.redirect('/newroute');
});
```

In this example, when a GET request is made to /oldroute, the server responds by redirecting the client to /newroute.

Combining Both Methods

In some scenarios, you might want to decide between sending a JSON response or redirecting based on certain conditions. Here's an example of how you can use both methods conditionally:

Example

```
app.get('/checklogin', (req, res) => {
  const isLoggedIn = false; // Example condition

  if (isLoggedIn) {
    res.json({ message: 'Welcome back!' });
  } else {
    res.redirect('/login');
  }
});
```

In this example, if the user is logged in, the server responds with a JSON message. If the user is not logged in, the server redirects the client to the login page.

Summary

json(): Used to send a JSON response, setting the ContentType to application/json.

redirect(): Used to redirect the client to a different URL, optionally specifying the HTTP status code.

Understanding and using these methods appropriately helps in creating robust and flexible Express.js applications.

Response Methods :render() and send(),

https://youtu.be/4JZxzyx9ggk

In the context of web frameworks, particularly in Node.js and Express.js, `res.render()` and `res.send()` are methods used to send responses to the client. Here's a detailed explanation of each:

`res.render()`

The `res.render()` method is used to render a view template and send the rendered HTML string to the client. It is typically used when you have a template engine set up in your Express.js application.

Syntax:

```
res.render(view [, locals] [, callback])
```

Parameters:

- view: The name of the view file to render (relative to the views directory).
- locals (optional): An object containing local variables to pass to the view template.
- callback (optional): A callback function that is called after the view is rendered.

Example:

```
app.get('/home', (req, res) => {
   res.render('home', { title: 'Home Page', user: req.user });
});
```

In this example, the `home` view template is rendered and the `title` and `user` variables are passed to it.

`res.send()`

The `res.send()` method is a versatile method for sending responses of various types to the client. It can send a response body in the form of a string, a buffer, or an object, which Express will automatically serialize to JSON.

Syntax:

```
res.send([body])
```

Parameters:

- body (optional): The body of the response. This can be a string, buffer, or object.

Example:

```
app.get('/json', (req, res) => {
  res.send({ message: 'Hello, World!' });
});

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

In these examples, the `/json` route sends a JSON response, while the `/text` route sends a plain text response.

Key Differences

- Purpose: `res.render()` is used for rendering templates and sending HTML to the client, while `res.send()` is used for sending various types of responses, including plain text, JSON, and HTML.
- Use Case: Use `res.render()` when you have a server-side template engine and want to render a view. Use `res.send()` when you want to send a response body directly.
- Flexibility: `res.send()` is more flexible in terms of the types of responses it can send, while `res.render()` is specifically for rendering views.

By understanding these methods, you can choose the appropriate one based on the type of response you need to send in your Express.js applications.

Response Methods : sendFile() and sendStatus(), https://youtu.be/EeVAFRMscr4

In Express.js, `sendFile()` and `sendStatus()` are two methods used to handle responses in different ways.

`sendFile()`

The `sendFile()` method is used to send a file to the client. This is useful when you want to serve static files like HTML, images, PDFs, etc. Here's how it works:

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

app.get('/file', (req, res) => {
  const filePath = path.join(__dirname, 'path/to/your/file.ext');
  res.sendFile(filePath, (err) => {
    if (err) {
      res.status(500).send('Error occurred while sending the file.');
    }
  });
});

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

- Parameters:
  - `path`: The absolute path of the file to be sent.
  - `options` (optional): Options for the file serving, such as `maxAge` for cache control.
  - `callback` (optional): A function to handle errors.

`sendStatus()`

The `sendStatus()` method is a shorthand for setting the HTTP status code and sending its string representation as the response body. It's useful for sending simple status codes without any additional content.

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

app.get('/status', (req, res) => {
  res.sendStatus(200); // This sends a response with status code 200 and the text "OK"
});

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

- Parameters:
  - `statusCode`: The HTTP status code to send.

Differences and Use Cases

- sendFile(): Use this when you need to send a file as a response. It's suitable for serving static files or any content stored as files on the server.
- sendStatus(): Use this for sending only the status code without any body content. It's useful for simple status updates, error handling, or endpoints that only need to acknowledge the request with a status code.

Examples

1. Sending an Image File:

```
app.get('/image', (req, res) => {
  res.sendFile(path.join(__dirname, 'public/images/picture.jpg'));
});
```

2. Sending a Status Code:

```
app.get('/not-found', (req, res) => {
  res.sendStatus(404); // Sends "Not Found"
});
```

3. Sending a File with Error Handling:

```
app.get('/report', (req, res) => {
  const reportPath = path.join(__dirname, 'reports/annual-report.pdf');
  res.sendFile(reportPath, (err) => {
    if (err) {
      res.sendStatus(500); // Internal Server Error
    }
  });
});
```

These methods help you manage different types of responses effectively in an Express.js application.

Lab Program-5,

https://youtu.be/nsew8YEDw6w

/*write a NodeJS program to accept a file name from the user , text from user ,if file exists append the text to the file , if not create a new file and add the text to it */

HTML (index.html)

Create an HTML form to get the file name and text from the user.

```
html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>File Writer</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 50px;
    }
    form {
      max-width: 500px;
      margin: auto;
    }
    label {
      display: block;
      margin: 10px 0 5px;
    }
    input, textarea {
      width: 100%;
      padding: 10px;
      margin-bottom: 10px;
      border: 1px solid ccc;
      border-radius: 4px;
    }
    button {
      padding: 10px 20px;
      background-color: 28a745;
      color: white;
      border: none;
      border-radius: 4px;
      cursor: pointer;
    }
    button:hover {
      background-color: 218838;
    }
  </style>
</head>
<body>
  <h1>File Writer</h1>
  <form action="/write-file" method="post">
    <label for="fileName">File Name</label>
    <input type="text" id="fileName" name="fileName" required>
    <label for="text">Text</label>
    <textarea id="text" name="text" rows="4" required></textarea>

    <button type="submit">Submit</button>
  </form>
</body>
</html>
```

Node.js (server.js)

Set up an Express server to handle the form submission and write or append the text to the specified file.

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

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

// Middleware to parse form data
app.use(bodyParser.urlencoded({ extended: true }));

// Serve the HTML file
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'index.html'));
```

```
});

// Handle form submission
app.post('/write-file', (req, res) => {
  const { fileName, text } = req.body;

  const filePath = path.join(__dirname, fileName);

  // Check if the file exists
  if (fs.existsSync(filePath)) {
    // If file exists, append the text
    fs.appendFile(filePath, text + '\n', (err) => {
      if (err) {
        console.error('An error occurred:', err);
        res.status(500).send('An error occurred while appending the text.');
      } else {
        res.send('Text appended to file.');
      }
    });
  } else {
    // If file doesn't exist, create a new file and write the text
    fs.writeFile(filePath, text + '\n', (err) => {
      if (err) {
        console.error('An error occurred:', err);
        res.status(500).send('An error occurred while creating the file.');
      } else {
        res.send('New file created and text written.');
      }
    });
  }
});

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

How to Run the Application

1. Save the HTML code to a file named index.html.
2. Save the Node.js code to a file named server.js.
3. Open a terminal or command prompt.
4. Navigate to the directory where the server.js file is saved.
5. Run the following commands to install the required dependencies and start the server:

```
npm init -y
npm install express body-parser
node server.js
```

6. Open your web browser and navigate to http://localhost:3000.
7. Fill out the form and submit it. The server will handle the file writing or appending based on the provided file name.