Handling Errors

Creating Logger Middleware

Logging is the process of creating logs to track important information in an application. Logging can be used to track errors, record request data, and provide insights for debugging.

Benefits of Logging

- Logging allows developers to analyze errors and fix them efficiently.
- Logging request data helps in understanding user behavior and identifying issues.
- Logs provide a historical record of events, aiding in troubleshooting and auditing.

Logger Middleware Implementation

- We import the `fs` module and `fs.promises` using `import`.
- The `log` function is created as an asynchronous function.
- Within the `log` function, we use `try-catch` to handle errors.
- The `writeFile` function is called to write the log data to a file.
- A timestamp is added to the log data using `new Date().toString()`.
- The log data is exported from the module.

Logger Middleware Usage

- The logger middleware is created as a function that takes `request`, `response`, and `next`.
- The `log` function is invoked within the logger middleware to log the request body.
- The `next` function is called to pass control to the next middleware in the pipeline.

logger.middleware.js file:

```
import fs from 'fs';

const fsPromise = fs.promises;

async function log(logData) {
  try {
```

```
    logData =
      new Date().toString() +
      '. Log Data: ' +
      logData;
    await fsPromise.writeFile('log.txt', logData);
  } catch (err) {
    console.log(err);
  }
}

const loggerMiddleware = async (
  req,
  res,
  next
) => {
  // 1. Log request body.
  await log(req.body);
  next();
};

export default loggerMiddleware;
```

Using Middleware Logger

We will use the logger middleware and test if the requests are being logged properly.

Applying Logger Middleware

- The logger middleware can be applied at different levels: application level, route level, or controller level.
- We apply the logger middleware at the application level using `server.use(loggerMiddleware)`.

Testing the Logger Middleware

- We use a tool like Postman to send requests and check if they are being logged.
- After sending a request, we check the logs to see if the timestamp and log data are present.
- Initially, the log data may display as `[object Object]` because we need to convert it to a string.

Enhancing the Logger Middleware

- We modify the logger middleware to convert the request body to a string and log the request URL.
- The `JSON.stringify()` method is used to convert the request body object to a string.
- The `appendFile` function is used instead of `writeFile` to preserve existing log data.
- A new line character (`\n`) is appended before each log entry for readability.

Modified **logger.middleware.js** file**:**

```
import fs from 'fs';
const fsPromise = fs.promises;

async function log(logData) {
  try {
    logData = `\n ${new Date().toString()} - ${logData}`;
    await fsPromise.appendFile(
      'log.txt',
      logData
    );
  } catch (err) {
    console.log(err);
  }
}

const loggerMiddleware = async (
  req,
  res,
  next
) => {
  // 1. Log request body.
  if (!req.url.includes('signin')) {
    const logData = `${
      req.url
    } - ${JSON.stringify(req.body)}`;
    await log(logData);
  }
  next();
```

```
};

export default loggerMiddleware;
```

Logging Best Practices

- We should avoid logging sensitive information such as passwords.
- We can skip logging specific routes or requests, like sign-in requests, to prevent logging sensitive data.
- Applying the logger middleware at the route level allows for more granular control over which requests are logged.

Using winston logger

Using the Winston library to implement a more effective logging system in a Node.js and Express API application.The current logging system uses the file system and FS promises to log requests from clients.

Logger Configuration

- The first step is to install the Winston library using `npm i winston`
- The logger is configured using Winston's `createLogger` method and an options object.
- The configuration options include setting the log level (e.g., info, error), log format (e.g., JSON), and log transport (e.g., file, console).

Changes in **logger.middleware.js**:

```
const logger = winston.createLogger({
  level:'info',
  format: winston.format.json(),
  defaultMeta: { service: 'request-logging' },
  transports:[
    new winston.transports.File({filename:'logs.txt'})
  ]
});
```

- The logger is called within the logger middleware to log incoming requests.

```
const loggerMiddleware = async (
  req,
  res,
  next
) => {
```

```
  // 1. Log request body.
  if (!req.url.includes('signin')) {
    const logData = `${
      req.url
    } - ${JSON.stringify(req.body)}`;
    logger.info(logData)
  }
  next();
};
```

Error Handling in Express

Proper error handling is an essential feature in any backend application. We will learn about handling exceptions and errors in an Express application.

Handling Errors in Controllers

- In JavaScript, we can use the `try-catch` block to handle exceptions.
- Instead of returning error messages from controllers, we can throw errors using the `throw` keyword.
- By throwing errors, we can catch them in the `catch` block and handle them accordingly.

Updating the Product Model and Controller code

- We update the product model code to throw errors instead of returning error messages.

```
if (!product) {
  throw new Error('Product not found');
}
```

- Inside controller the `try` block, we call the function that may throw an error, and in case of an error, it will be caught in the `catch` block.

```
rateProduct(req, res) {
    console.log(req.query);
    const userID = req.query.userID;
    const productID = req.query.productID;
    const rating = req.query.rating;
    try {
      ProductModel.rateProduct(
        userID,
        productID,
        rating
      );
    } catch (err) {
      return res.status(400).send(err.message);
    }
    return res
      .status(200)
      .send('Rating has been added');
  }
```

- The `catch` block can return the error message back to the client.

Handling Different Types of Errors

- We can throw different types of errors depending on the situation.
- For example, if a product is not found, we can throw a new error with a "Product not found" message.
- We can also handle errors in asynchronous operations using `try-catch` blocks.

Improving Error Messages

- We should improve error messages to make them more meaningful to the user.
- In case of a bad request or invalid input, we return the appropriate error message.
- The error message should provide useful information to the user without exposing sensitive details.

Application Level Error Handling

- We learned how to handle errors in Express using try-catch blocks and throwing user-defined errors.
- However, there are system-level errors that can occur in our application and need to be handled differently.
- Handling errors at the application level ensures consistent error handling and provides a better response to clients.

Differentiating User-Defined Errors and System Errors

- User-defined errors are errors that we define in our code using the `throw` keyword.
- System errors are exceptions or errors that are not intentionally thrown by us but occur due to issues in the application.

Need for Application Level Error Handling

- Instead of writing try-catch blocks in every controller function, it's better to have a centralized error handler at the application level.
- An error handler middleware can catch and handle any unhandled errors in the application.
- This error handler middleware provides a more controlled and meaningful response to clients.

Implementing Error Handler Middleware

- Express provides a default error handler middleware, but we can customize it to meet our requirements.
- By setting up an error handler middleware, we can catch and process errors in a centralized manner.

Customizing Error Messages

- The default error handler middleware returns the entire stack trace, which is not suitable for client consumption.
- We can customize the error handler middleware to return more meaningful error messages to clients.
- The error handler middleware should focus on providing a clear and understandable response to the client, without exposing internal details of the application.

Logging Errors

- It is important to log errors for debugging and troubleshooting purposes.
- We can integrate our existing logger middleware with the error handler middleware to log errors.
- This ensures that we have a record of errors and can investigate and address them effectively.

```
// Error handler middleware in server.js
server.use((err, req, res, next) => {
  console.log(err);
  res
    .status(503)
    .send(
      'Something went wrong, please try later'
    );
```

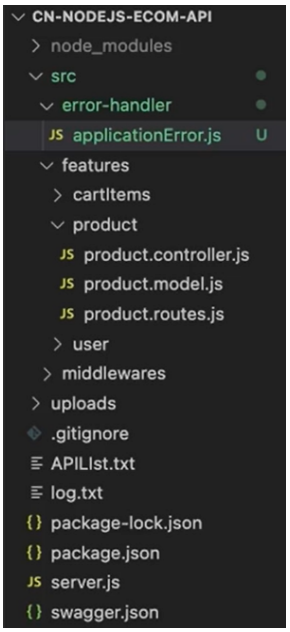
```
});
```

Customizing Error Handling Middleware

Here we will focus on customizing our error class and error handler to work with different types of error messages and status codes.

Extending the Error Class

- To customize our error handling, we create a new class called `ApplicationError` that extends the JavaScript error class.




```
export class ApplicationError extends Error {
  constructor(message, code) {
    super(message);
    this.code = code;
  }
}
```

- This custom error class allows us to add additional properties, such as a status code, to provide more information about the error.

Customizing Error Messages and Status Codes

- By extending the error class, we can specify both the error message and the status code when throwing an error.
- For user-defined errors, we can set the appropriate status code (e.g., 400 or 404) to indicate the type of error.

Modified **product.model.js** file

```
  static rateProduct(userID, productID, rating) {
    // 1. Validate user and product
    const user = UserModel.getAll().find(
      (u) => u.id == userID
    );
```

```
    if (!user) {
      // user-defined error.
      throw new ApplicationError(
        'User not found',
        404
      );
    }

    // Validate Product
    const product = products.find(
      (p) => p.id == productID
    );
    if (!product) {
      throw new ApplicationError(
        'Product not found',
        400
      );
    }
```

- Server errors, which are not user-defined, can use a general status code of 500.
- In the error handler middleware, we can check if the error is an instance of `ApplicationError` and return the corresponding status code and message.
- For other errors that are not instances of `ApplicationError`, we can send a generic 500 status code and a generic error message.

Modified **server.js** file:

```
// Error handler middleware
server.use((err, req, res, next) => {
  console.log(err);
  if (err instanceof ApplicationError) {
    res.status(err.code).send(err.message);
  }

  // server errors.
  res
    .status(500)
    .send(
      'Something went wrong, please try later'
    );
});
```

Logging Errors

- It's important to log errors for debugging and troubleshooting purposes.
- While we don't need to log every error in the error handler middleware, we should log the server errors for further investigation and resolution.

Testing Customized Error Handling

- We can test our customized error handling by intentionally causing errors and observing the error messages and status codes returned.
- By providing specific status codes and messages, we can enhance the user experience and provide meaningful error responses.

Summarizing Project

The project covered various topics, from creating the folder structure to implementing different functionalities in the application.

1. Folder Structure

- The project followed a feature-based folder structure, with separate folders for different features such as controllers, models, and rules.
- The middlewares folder housed all the middleware functions, and server configuration was handled separately.

2. API Development

- Multiple APIs were created for different features, including products, user authentication and registration, and cart items.
- The project demonstrated the use of different HTTP methods like GET, POST, PUT, and DELETE, along with the OPTIONS method for pre-flight requests.
- Request and response headers were utilized, and middlewares played a crucial role in authentication, logging, request handling, and body parsing.

3. Parameters and Query Handling

- Parameters in routes, such as retrieving a product by ID, and query parameters, such as product ID, user ID, and rating in the rate API, were properly handled in Express.

4. Postman

- Postman was highlighted as a valuable tool for testing APIs, especially in cases where direct access to the UI might not be available.
- It allows API developers to test and document their APIs efficiently, enabling effective communication with consumers.

5. Application Security

- Two popular mechanisms for securing APIs were explored: basic authentication and token authentication using JWT.
- Token authentication with JWT was recommended for its security and user-friendliness.

6. API Documentation with Swagger

- Swagger was introduced as a powerful tool for documenting APIs.
- It helps consumers understand the available APIs, their expectations, and how to consume them.
- Swagger also provides an interface to test APIs directly from its UI.

7. Cross-Origin Request Handling

- Cross-origin request handling was achieved through the use of a middleware and an NPM library with the same name.
- The CORS policy was configured to define allowed origins, methods, and headers for accessing the application.

8. Logging

- The importance of logging in server applications was emphasized.
- Logging was implemented for both requests and errors, aiding in debugging and troubleshooting.
- Server errors were specifically highlighted as crucial for logging and future analysis.

9. Error Handling

- Error handling was discussed as an essential part of the application to enhance the consumer experience.
- Sending appropriate error messages instead of stack traces improves user understanding and protects against potential misuse.
- Logging the stack traces in a secure location, such as a database or text files, was recommended.

Best Practices

Following these practices enhances the consumer experience and improves developer efficiency.

1. Using Correct Status Codes

- Using appropriate status codes provides accurate information about the request's status.

- Incorrect status codes can confuse users and lead to incorrect details being provided.
- Choosing the right status codes creates a more user-friendly API.

2. Error Handling

- Proper error handling is crucial for a good consumer experience.
- Providing correct error messages prevents the exposure of internal details to external clients.
- Handling errors correctly ensures that error messages are meaningful and helpful for front-end developers.

3. Validating Data

- Data validation is essential to ensure the integrity and security of the application.
- Validating data received from consumers helps protect against malicious inputs, such as SQL injection.
- Express Validator middleware can be used to validate data before processing requests.

4. Using Correct Request Methods

- Adhering to the principles of REST, the correct request methods should be used.
- GET is for retrieval, POST is for adding resources, and other methods have specific purposes.
- Using the correct methods ensures consistent and predictable behavior for servers and clients.

5. Securing APIs

- Security is crucial, especially when authentication is required.
- Basic authentication, JWT authentication, and third-party authentication options can be implemented.
- Measures should be taken to prevent unauthorized access to the application.

6. Using Correct Naming Conventions

- Properly naming URLs and resources is essential for clarity and consistency.
- Nouns should be used to represent resources, while verbs should be avoided.
- Forward slashes can indicate hierarchical relationships, and hyphens can enhance readability.
- Lowercase letters should be used in URLs, and CRUD function names should be avoided.
- Query components can be used to filter URL collections.

Summarising it

Let’s summarise what we have learned in this module:

- Created and utilized logger middleware to handle logging in our application effectively.
- Implemented the winston logger in our application, which is a popular logging library for Node.js and Express.
- Covered error handling in Express, where we learned how to handle errors that occur during the application's execution.
- Applied application-level error handling to deal with errors at a higher level and ensure smoother application flow.
- Summarized the project's main aspects and achievements.
- Explored best practices that can enhance the consumer experience and improve overall developer efficiency during the development process.

Some Additional Resources:

- Creating a logging middleware in Expressjs
- Error Handling in Express
- Winston Logger Ultimate Tutorial