Exploration — Express Middleware

Introduction

We have already used Express for writing our server programs.

- To define routes, we used the `app.METHOD` API.
- To serve static files, we added the following statement to our server programs:
    - `app.use(express.static('public'));`
- When our server program needed to process forms, we added the following statement to our program
    - `app.use(express.urlencoded({extended: true}));`

But we have not explained the details of `app.use()` API and how using this API makes additional functionality available to our server programs. In this exploration, we will explain this topic by studying the concept of **middleware**. We will learn how we can add functionality to our Express applications by using middleware that is already available and by writing new middleware ourselves.

What is Middleware?

A middleware is simply a function that Express applies to an HTTP request in our Express programs. We can think of an Express program as a pipeline of middleware functions which are applied to an HTTP request.

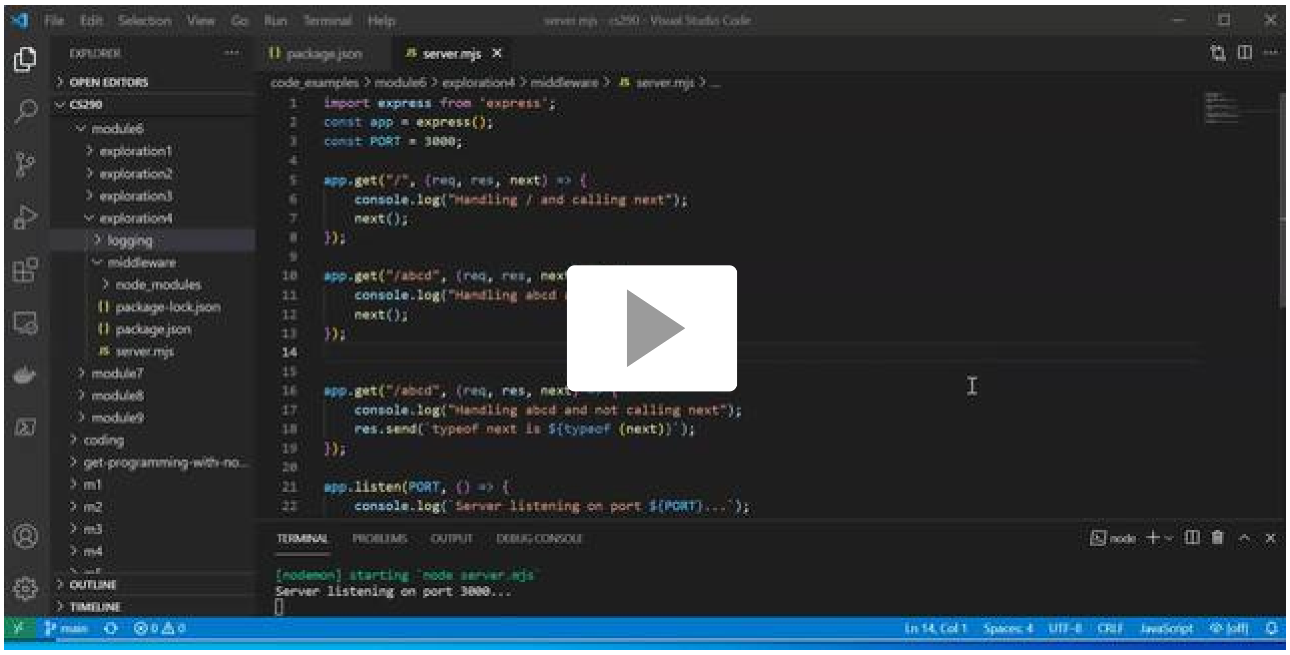
- Most middleware functions take three arguments:
    - A request object,
    - A response object, and
    - A `next()` function.
- Another form of middleware functions, called Error-handling middleware, takes four arguments:
    - An error object,
    - A request object,
    - A response object, and
    - A `next()` function.
- Each middleware function in an Express app is associated with a path (or route). A middleware function applies to an HTTP request only if that function's path matches the URL of the request.
- The middleware functions with routes matching a request's URL are applied in the exact order of how they appear in the code of our Express app, thus setting up a pipeline of functions.
- A middleware function can make changes to the request and response objects passed to it as arguments.

- A middleware function can pass the request to the next middleware function in the pipeline by calling the `next()` function that is provided to it as an argument.
- The pipeline of middleware functions terminates when:
  - Either all the middleware functions matching the request have been applied to the request,
  - Or a middleware function does not call `next()` and thus does not pass the request to the next function in the pipeline. Typically, this is done by a middleware function which sends back the HTTP response.

That's a lot of concepts! Let us look at specific Express API and example programs that use this API to help us digest these concepts.

app.METHOD

We have already used app.METHOD functions to add route handlers for HTTP requests. The route handlers we defined are examples of Express middleware. However, when defining the route handlers we defined functions that took two parameters, `request` and `response`, rather than three parameters. The reason for this is that typically a route handler sends back a response and does not invoke any middleware function that might follow it in the pipeline. If we were to define a route handler with a third parameter, the `next()` function will be available in the body of our route handler.



```
import express from 'express';
const app = express();
const PORT = 3000;

app.get("/", (req, res, next) => {
    console.log("Handling / and calling next");
    next();
});

app.get("/abcd", (req, res, next
    console.log("Handling abcd
    next();
});


app.get("/abcd", (req, res, next
    console.log("Handling abcd and not calling next");
    res.send(`typeof next is ${typeof (next)}`);
});

app.listen(PORT, () => {
    console.log(`Server listening on port ${PORT}...`);
```

```
[nodemon] starting `node server.mjs`
Server listening on port 3000...
```

0:00 / 6:55 1x

Example: Route Handler with next

If we add the following route handler in our server program, the response will contain the text "typeof next is function".

```
  app.get("/", (req, res, next) => {
      res.send(`typeof next is ${typeof (next)}`);
  });
```

Exercise: Using next in a Route Handler

If our server program contained only the following 3 routes and no other middleware, what will be printed on the console for a request for the URL `/abcd`? Explain your answer.

```
// route handler 1
app.get("/", (req, res, next) => {
    console.log("Handling / and calling next");
    next();
});

// route handler 2
app.get("/abcd", (req, res, next) => {
    console.log("Handling abcd and calling next");
    next();
});

// route handler 3
app.get("/abcd", (req, res, next) => {
    console.log("Handling abcd and not calling next");
    res.send(`typeof next is ${typeof (next)}`);
});
```

Exercise: Order of Route Handlers

If our server program contained only the following 2 routes and no other middleware, what will be printed on the console for a request for the URL `/abcd`? Explain your answer.

```
// route handler 1
app.get("/abcd", (req, res, next) => {
    console.log("Handling abcd and not calling next");
    res.send(`typeof next is ${typeof (next)}`);
});

// route handler 2
app.get("/abcd", (req, res, next) => {
    console.log("Handling abcd and calling next");
    next();
});
```

Calling next() and Sending the Response

What happens if a middleware function is executed, it sends back the response and also calls the `next()` function? In this case, the next middleware function in the pipeline is invoked. However, any responses sent from this function or any later function in the pipeline will be ignored.

Not Calling next() and Not Sending the Response

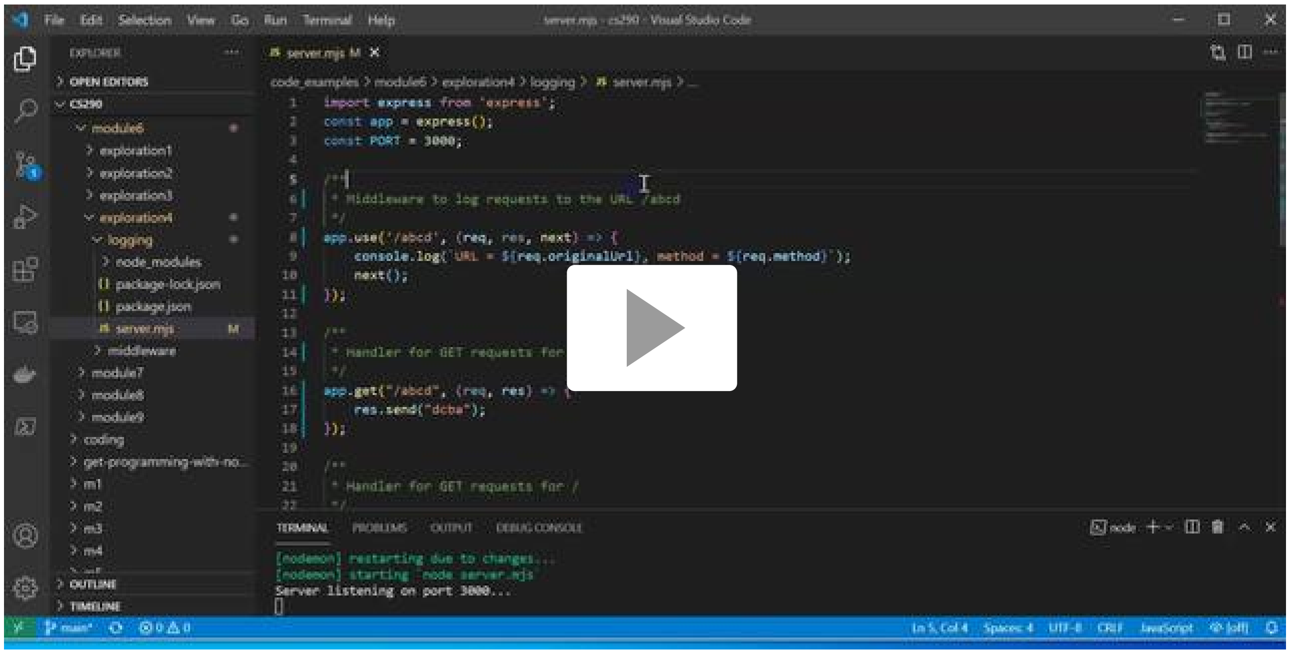
What happens if a middleware function is executed, but it does not call `next()` and does not send back a response? In this case, because `next()` has not been called the pipeline terminates and no other middleware function will be executed. However, no response has been sent back to the client program which sent the request. This will cause the client to ultimately timeout and report an error, e.g., a network error.

app.use()

Our other use of Express middleware has been by calling the `app.use()` API to register the middleware function to execute. For example, we added functionality to serve static files with the following statement: `app.use(express.static('public'));`. We note a few things about this statement:

1. We are calling the function `express.static()` in the above statement and passing its return value as an argument to `app.use()`.
    - `express.static()` is not the middleware function.
    - But `express.static()` returns another function. This is the middleware function that is passed to `app.use()`.
2. The call to `app.use()` does not specify an argument for the path for which this middleware function will be applied.
    - This is because `app.use()` has the default value `/` for the path argument and the middleware function registered with the default value can be applied to every request to the server program.
    - If we want a middleware function registered using `app.use()` to apply only to specific paths, we can supply the desired path as an argument to `app.use()`.
3. The middleware function will apply to request regardless of the HTTP method as long as the path matches the URL of the request.

Here is the **server.mjs (https://canvas.oregonstate.edu/courses/1879154/files/93831868?wrap=1)** (https://canvas.oregonstate.edu/courses/1879154/files/93831868/download?download_frd=1) file used in the following video and the one after it.



```
import express from 'express';
const app = express();
const PORT = 3000;

/**
 * Middleware to log requests to the URL /abcd
 */
app.use('/abcd', (req, res, next) => {
    console.log(`URL = ${req.originalUrl}, method = ${req.method}`);
    next();
});

/**
 * Handler for GET requests for
 */
app.get("/abcd", (req, res) => {
    res.send("dcba");
});

/**
 * Handler for GET requests for /
 */
```

```
[nodemon] restarting due to changes...
[nodemon] starting `node server.mjs`
Server listening on port 3000...
```

0:00 / 8:42

Example: Defining and Registering Our Own Middleware Function

In the following example, we define a middleware function to log the URL and method for all requests for the URL `/abcd`.

```
app.use('/abcd', (req, res, next) => {
    console.log(`URL = ${req.originalUrl}, method = ${req.method}`);
    next();
})
```

Note that the statements to register this function must appear before any other middleware function that matches this URL but does not call `next()`. Otherwise, our logging middleware function will never be invoked because the pipeline will terminate before its execution.

Example: Using Named Middleware Functions

We defined the middleware function to log the URL and method as an anonymous function within the route handler for `'/abcd'`. It is also possible to use named middleware functions in the route handlers and with `app.use` API. In the following example, we define the previously anonymous logging middleware function as a named function.

```
function logUrls(req, res, next) {
    console.log(`URL = ${req.originalUrl}, method = ${req.method}`);
    next();
}
```

We can then use the function `logUrls()` in any route handler we want. In fact, a route handler can be passed multiple middleware functions and each one of these will be added to the pipeline of middleware functions for that route. For example, the following route handler for the URL `'/xyz'` is passed two functions, the named function `logUrls()` and an anonymous function that sends back a response. Since the function `logUrls()` calls the function `next()`, therefore after `logUrls()` completes its execution the anonymous function will be called and a response will be sent back to the user,

```
app.use('/xyz', logUrls, (req, res, next) => {
    res.send("Hello from xyz!");
})
```

Error-handling Middleware

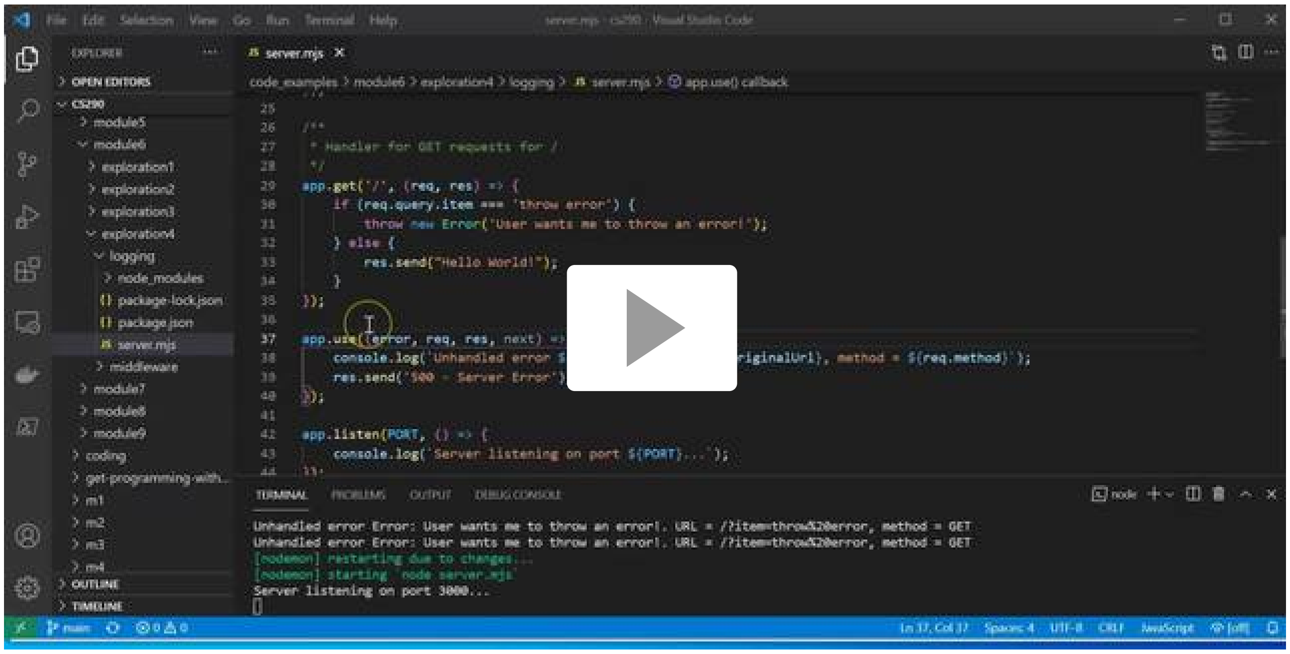
If a middleware function takes four parameters it becomes error-handling middleware. Error handling middleware should be placed after all the route handlers and other non-error handling middleware.

- The four parameters of error-handling middleware are:
  1. The `error` object,
  2. The `request` object,
  3. The `response` object, and
  4. The `next` function.
- If an error handling middleware does not call `next()`, it should return a response. Otherwise the client will hang waiting for the response and will eventually timeout.
- Note that even if we do not call `next()` in the error-handling middleware, we still must declare the function to take four arguments. Otherwise, Express will treat it as regular middleware and pass it `request`, `response` and `next()`.

Example: Error-handling Middleware

In the following example, we register an error handling middleware that will be invoked whenever any earlier middleware function throws an uncaught exception.

```
app.use((error, req, res, next) => {
    console.log(`Unhandled error ${error}. URL = ${req.originalUrl}, method = ${req.method}`);
    res.send('500 - Server Error');
});
```


server.mjs
code_examples > module6 > exploration4 > logging > server.mjs > app.use() callback
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
/**
 * Handler for GET requests for /
 */
app.get('/', (req, res) => {
    if (req.query.item === 'throw error') {
        throw new Error('User wants me to throw an error!');
    } else {
        res.send("Hello World!");
    }
});

app.use((error, req, res, next) => [illegible]
    console.log(`Unhandled error $[illegible]riginalUrl}, method = ${req.method}`);
    res.send('500 - Server Error')[illegible]
});

app.listen(PORT, () => {
    console.log(`Server listening on port ${PORT}...`);
});
```

```
Unhandled error Error: User wants me to throw an error!. URL = /?item=throw%20error, method = GET
Unhandled error Error: User wants me to throw an error!. URL = /?item=throw%20error, method = GET
[nodemon] restarting due to changes...
[nodemon] starting `node server.mjs`
Server listening on port 3000...
```

0:00 / 4:59

Summary

In this exploration, we studied Express middleware. The Express website lists many Express **middleware modules (https://expressjs.com/en/resources/middleware.html)**. So chances are that you will find a middleware module that implements the functionality you need. However, as we saw, it is also fairly simple to write and use our own middleware functions.

Additional Resources

- Express documentation is a good reference to learn more about middleware. Look at the docs on **using middleware (https://expressjs.com/en/guide/using-middleware.html)** and on **writing middleware (https://expressjs.com/en/guide/writing-middleware.html)**.
- For a discussion of error handling in Express **see this page (https://expressjs.com/en/guide/error-handling.html)**.