Agenda

- Express Middlewares
- Environment Variables
- Error Handling with Express

Express Middlewares

What is Express Middleware?

Middleware in Express are functions that come into play **after the server receives the request and before the response is sent to the client**. The are arranged in a chain and are called in sequence.

We can use middleware functions for different types of processing tasks required for fulfilling the request like database querying, making API call preparing the response, etc, and finally calling the next middleware function in the chain.

Middleware functions take three arguments: the request object (`request`), the response object (`response`), and optionally the `next()` middlewa function:

```
const express = require('express');
const app = express();
function middlewareFunction(request, response, next){
  ...
  next()
}

app.use(middlewareFunction)
```

An exception to this rule is error handling middleware (we will get to it in future) which takes an error object as the fourth parameter. W call `app.use()` to add a middleware function to our Express application.

Under the hood, when we call `app.use()` , the Express framework adds our middleware function to its internal middleware stack. Express execute middleware in the order they are added, so if we make the calls in this order:

```
app.use(function1)
app.use(function2)
```

Express will first execute `function1` and then `function2` .

Middleware functions in Express are of the following types:

- Application-level middleware which runs for all routes in an `app` object
- Router level middleware which runs for all routes in a router object
- Built-in middleware provided by Express like `express.static` , `express.json` , `express.urlencoded`
- Error handling middleware for handling errors
- Third-party middleware maintained by the community

We will see examples of each of these types in the subsequent sections.

We need to first set up a Node.js project for running our examples of using middleware functions in Express.

Let us create a folder and initialize a Node.js project under it by running the `npm init` command:

```
mkdir storefront
cd storefront
npm init -y
```

Running these commands will create a Node.js project containing a `package.json` file.

We will next install the Express framework using the `npm install` command as shown below:

```
npm install express --save
```

When we run this command, it will install the Express framework and also add it as a dependency in our `package.json` file.

We will now create a file named `index.js` and open the project folder in our favorite code editor. We are using Visual Studio Code as our source-coc editor.

Let us now add the following lines of code to `index.js` for running a simple HTTP server:

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

const app = express();

// Route for handling get request for path /
app.get('/', (request, response) => {
    response.send('response for GET request');
})

// Route for handling post request for path /products
app.post('/products', (request, response) => {
  ...
  response.json(...)
})

// start the server
app.listen(3000,
   () => console.log('Server listening on port 3000.'))
```

In this code snippet, we are importing the `express` module and then calling the `listen()` function on the `app` handle to start our server.

We have also defined two routes which will accept the requests at URLs: `/` and `/products` .

We can run our application with the `node` command:

```
node index.js
```

This will start a server that will listen for requests in port `3000` . We will now add middleware functions to this application in the following sections.

Using Express' Built-in Middleware

Built-in middleware functions are bundled with Express so we do not need to install any additional modules for using them.

Express provides the following Built-in middleware functions:

Let us see some examples of their use.

Using `express.static` for Serving Static Assets

We use the `express.static` built-in middleware function to serve static files such as images, CSS files, and JavaScript files. Here is an example using `express.static` to serve our HTML and image files:

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

const app = express();
app.use(express.static('images'))
app.use(express.static('htmls'))

app.get('product', (request, response)=>{
  response.sendFile("productsample.html")
})
```

Here we have defined two static paths named `images` and `htmls` to represent two folders of the same name in our root directory. We have als defined multiple static assets directories by calling the `express.static()` middleware function multiple times.

Our root directory structure looks like this:

```
.
├── htmls
│   └── productsample.html
├── images
│   └── sample.jpg
```

```
├── index.js
├── node_modules
```

Express looks for the files in the order in which we set the static directories with the `express.static` middleware function.

In our example, we have defined the `images` directory before `htmls`. So Express will look for the file: `productsample.html` the `images` directory first. If the file is not found in the `images` directory, Express looks for the file in the `htmls` directory.

Next we have defined a route with url `product` to serve the static HTML file `productsample.html`. The HTML file contains an image referred on with the image name `sample.jpg`:

```
<html>
<body>
    <h2>My sample product page</h2>
    <img src="sample.jpg" alt="sample"></img>
</body>
</html>
```

Express looks up the files relative to the static directory, so the name of the static directory is not part of the URL.

**Using `express.json` for Parsing JSON Payloads

We use the `express.json` built-in middleware function to JSON content received from the incoming requests.

Let us suppose the route with URL `/products` in our Express application accepts `product` data from the `request` object in JSON format. So w will use Express' built-in middleware `express.json` for parsing the incoming JSON payload and attach it to our `router` object as shown in this coc snippet:

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

const app = express();

// Attach the express.json middleware to route "/products"
app.use('/products', express.json({ limit: 100 }))

// handle post request for path /products
app.post('/products', (request, response) => {
...
...
  response.json(...)
})
```

Here we are attaching the `express.json` middleware by calling the `use()` function on the `app` object. We have also configured a maximum siz of `100` bytes for the JSON request.

We have used a slightly different signature of the `use()` function than the signature of the function used before. The `use()` function invoked c the `app` object here takes the URL of the route: `/products` to which the middleware function will get attached, as the first parameter. Due to this, th middleware function will be called only for this route.

Now we can extract the fields from the JSON payload sent in the request body as shown in this route definition:

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

const app = express()

// Attach the express.json middleware to route "/products"
app.use('/products', express.json({ limit: 100 }))

// handle post request for path /products
app.post('/products', (request, response) => {
  const products = []

  // sample JSON request
  // {"name":"furniture", "brand":"century", "price":1067.67}

  // JSON payload is parsed to extract
  // the fields name, brand, and category
  // Extract name of product
  const name = request.body.name

  // Extract brand of product
  const brand = request.body.brand

  // Extract category of product
  const category = request.body.category

  console.log(name + " " + brand + " " + category)

...
...
  response.json(...)
})
```

Here we are extracting the contents of the JSON request by calling `request.body.FIELD_NAME` before using those fields for adding a new `product`

Similarly we can use express' built-in middleware `express.urlencoded()` to process URL encoded fields submitted through a HTTP form object:

```
app.use(express.urlencoded({ extended: false }));
```

Then we can use the same code for extracting the fields as we had used before for extracting the fields from a JSON payload.

Adding a Middleware Function to a Route

Let us now see how to create a middleware function of our own.

As an example, let us check for the presence of JSON content in the HTTP POST request body before allowing any further processing and send back a error response if the request body does not contain JSON content.

Our middleware function for checking for the presence of JSON content looks like this:

```
const requireJsonContent = (request, response, next) => {
  if (request.headers['content-type'] !== 'application/json') {
      response.status(400).send('Server requires application/json')
  } else {
    next()
  }
}
```

Here we are checking the value of the `content-type` header in the request. If the value of the `content-type` header does n match `application/json`, we are sending back an error response with status `400` accompanied by an error message thereby ending the reques response cycle.

Otherwise, if the `content-type` header is `application/json`, the `next()` function is invoked to call the subsequent middleware present in th chain.

Next we will add the middleware function: `requireJsonContent` to our desired route like this:

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

const app = express()

// handle post request for path /products
app.post('/products', requireJsonContent, (request, response) => {
  ...
  ...
  response.json(...)
})
```

We can also attach more than one middleware function to a route to apply multiple stages of processing.

Our route with multiple middleware functions attached will look like this:

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

// handle post request for path /products
app.post('/products',

  // first function in the chain will check for JSON content
  requireJsonContent,

  // second function will check for valid product category
  // in the request if the first function detects JSON
  (request, response) => {

     // Allow to add only products in the category "Electronics"
     const category = request.body.category
     if(category != "Electronics") {
      response.status(400).send('Server requires application/json')
     } else {
        next()
     }
  ...
  ...
  // add the product and return a response in JSON
  response.json(
    {productID: "12345",
    result: "success")}
  );
```

Here we have two middleware functions attached to the route with route path `/products` .

The first middleware function `requireJsonContent()` will pass the control to the next function in the chain if the `content-type` header in the HTT request contains `application/json` .

The second middleware function extracts the `category` field from the JSON request and sends back an error response if the value the `category` field is not `Electronics` .

Otherwise, it calls the `next()` function to process the request further which adds the product to a database for example, and sends back a response JSON format to the caller.

We could have also attached our middleware function by using the `use()` function of the `app` object as shown below:

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

const app = express()

// first function in the chain will check for JSON content
app.use('/products', requireJsonContent)

// second function will check for valid product category
// in the request if the first function detects JSON
app.use('/products',  (request, response) => {

     // Allow to add only products in the category "Electronics"
     const category = request.body.category
     if(category != "Electronics") {
      response.status(400).send('Server requires application/json')
     } else {
        next()
     }
   })

// handle post request for path /products
app.post('/products',
  (request, response) => {

  ...
  ...
  response.json(
      {productID: "12345",
      result: "success"})
    })
```

**Understanding The `next()` Function

The `next()` function is a function in the Express router that, when invoked, executes the next middleware in the middleware stack.

If the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware functio Otherwise, the request will be left hanging.

When we have multiple middleware functions, we need to ensure that each of our middleware functions either calls the `next()` function or sends ba a response. Express will not throw an error if our middleware does not call the `next()` function and will simply hang.

The `next()` function could be named anything, but by convention it is always named “next”.

Adding a Middleware Function to All Requests

We might also want to perform some common processing for all the routes and specify them in one place instead of repeating them for all the rou definitions. Examples of common processing are authentication, logging, common validations, etc.

Let us suppose we want to print the HTTP method (get, post, etc.) and the URL of every request sent to the Express application. Our middlewa function for printing this information will look like this:

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

const app = express();

const requestLogger = (request, response, next) => {
    console.log(`${request.method} url:: ${request.url}`);
    next()
}

app.use(requestLogger)
```

This middleware function: `requestLogger` accesses the `method` and `url` fields from the `request` object to print the request URL along with t HTTP method to the console.

For applying the middleware function to all routes, we will attach the function to the `app` object that represents the `express()` function.

Since we have attached this function to the `app` object, it will get called for every call to the express application. Now when w visit `http://localhost:3000` or any other route in this application, we can see the HTTP method and URL of the incoming request object in t terminal window.

Using Third-Party Middleware

We can also use third-party middleware to add functionality built by the community to our Express applications. These are usually available as np modules which we install by running the `npm install` command in our terminal window. The following example illustrates installing and loading third-party middleware named `Morgan` which is an HTTP request logging middleware for Node.js.

```
npm install morgan
```

After installing the module containing the third-party middleware, we need to load the middleware function in our Express application as shown below:

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

const app = express()

app.use(morgan('tiny'))
```

Here we are loading the middleware function `morgan` by calling `require()` and then attaching the function to our routes with the `use()` method the `app` instance.

Learn more more about morgan here.

Environment Variables

Environment variables are a fundamental part of developing with Node.js, allowing your app to behave differently based on the environment you wa them to run in. Wherever your app needs configuration, you use environment variables. And they're so simple, they're beautiful!

Why?

If you care about making your app run on any computer or cloud (aka your environments), then you should use them. **Why?** Because they externaliz all environment specific aspects of your app and keep your app encapsulated. Now you can run your app anywhere by modifying the environme variables without changing your code and without rebuilding it!

When?

OK, so now you ask, when you should use them. In short, **any place in your code that will change based on the environment**. When you see thes situations, use environment variables for anything you need to change or configure.

Here are some specific examples of common scenarios when you should consider using environment variables.

- Which HTTP port to listen on
- What path and folder your files are located in, that you want to serve
- Pointing to a development, staging, test, or production database

Other examples might be URLs to server resources, CDNs for testing vs. production, and even a marker to label your app in the UI by the environment lives in.

Let's explore how you can use environment variables in Node.js code.

Using Environment Variables

You may be setting a port number for an Express server. Often the port in a different environment (e.g.; staging, testing, production) may have to b changed based on policies and to avoid conflicts with other apps. As a developer, you shouldn't care about this, and really, you don't need to. Here how you can use an environment variable in code to grab the port.

```
// server.js
const port = process.env.PORT;
console.log(`Your port is ${port}`);
```

Go ahead and try this. Create an empty folder named `env-playground` . Then create a file named `server.js` and add the code above to it. No when you execute node server.js you should see a message that says "Your port is undefined".

Your environment variable isn't there because we need to pass them in. Let's consider some ways we can fix this.

1. using the command line
2. using a `.env` file

Command Line:

The simplest way to pass the port into your code is to use it from the command line. Indicate the name of the variable, followed by the equal sign, ar then the value. Then invoke your Node.js app.

```
PORT=8626 node server.js
```

You will see the port displayed in the message like this "Your port is 8626".You can repeat this pattern and add other variables too. Here is an example passing in two environment variables.

```
PORT=8626 NODE_ENV=development node server.js
```

Follow the pattern of environment variable name followed by the equal sign followed by the value. This is easy to do, but also far too easy to make typing mistake. Which leads to the next option.

Less Mess with a .env File:

Once you define several of these, the next thought that may cross your mind is how you can manage them, so they don't become a maintenanc nightmare. Imagine several of these for database connectivity and ports and keys. This doesn't scale well when you type them all on one line. And the could be private information such as keys, usernames, passwords, and other secrets.

Running them from a command line is convenient, sure. But it has its drawbacks:

1. there is no good place to see the list of variables

2. it's far too easy to make a typing mistake from the command line

3. it's not ideal to remember all of the variables and their values

4. even with npm scripts, you still have to keep them current

A popular solution to how you can organize and maintain your environment variables is to use a `.env` file. I really like this technique as it makes super easy to have one place where I can quickly read and modify them.

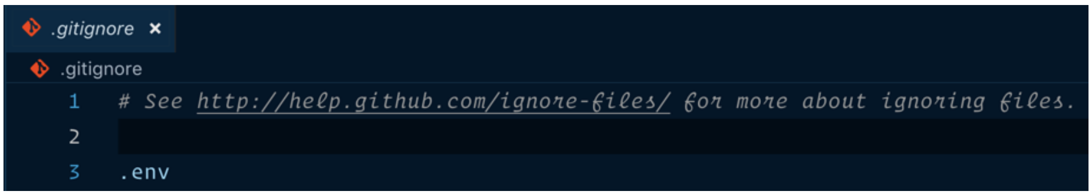
Create the `.env` file in the root of your app and add your variables and values to it.

```
NODE_ENV=development
PORT=8626
# Set your database/API connection information here
API_KEY=**************************
API_URL=**************************
```

Remember Your .gitignore File

A `.env` file is a great way to see all of your environment variables in one place. Just be sure not to put them into source control. Otherwise, your histo will contain references to your secrets!

Create a `.gitignore` file (or edit your existing one, if you have one already) and add `.env` to it, as shown in the following imag The `.gitignore` file tells source control to ignore the files (or file patterns) you list.



```
# See http://help.github.com/ignore-files/ for more about ignoring files.

.env
```

You can add a file to your `.gitignore` file by using the command palette in Visual Studio Code (VS Code). Follow these steps:

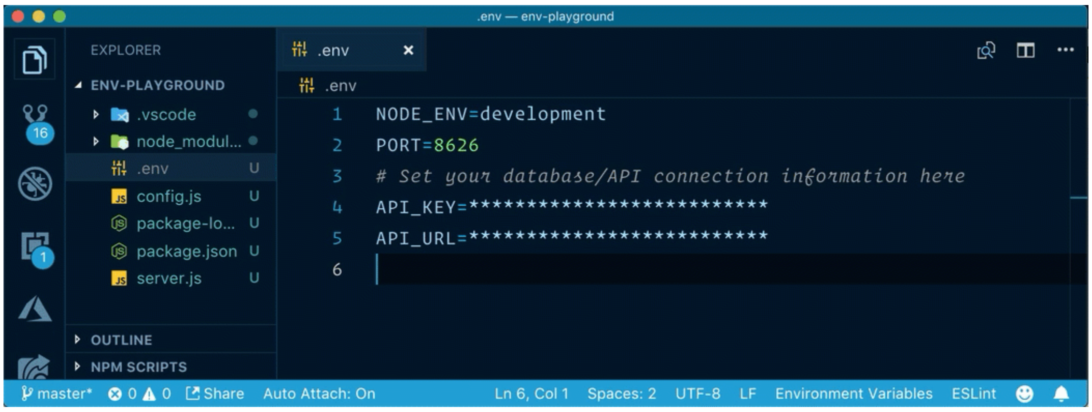
1. Open the file you want to add to the `.gitignore` in VS Code

2. Open the Command Palette with `CMD` + `SHIFT` + `P` on a Mac or `CTRL` + `SHIFT` + `P` on Windows

3. Type `ignore`

4. Select "Git: Add file to `.gitignore` from the menu"

This will add the name of the current file you have open to the `.gitignore` file. If you have not created a `.gitignore` file, it will create it for you!



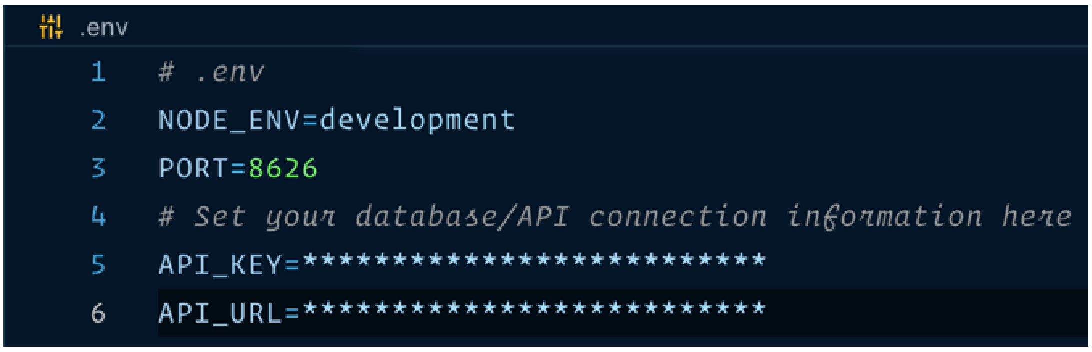
```
NODE_ENV=development
PORT=8626
# Set your database/API connection information here
API_KEY=**************************
API_URL=**************************

```

Syntax Highlighting for Your .env File

If you use VS Code you'll want to add the dotenv extension. This lights up the contents of your `.env` file with syntax highlighting and just plain old doe it easier to work with the variables inside of a `.env` file.

Here is a glimpse of the file in VS Code with the dotenv extension installed.



```
.env
# .env
NODE_ENV=development
PORT=8626
# Set your database/API connection information here
API_KEY=**************************
API_URL=**************************
```

Reading the .env File:

Right about now you're probably thinking that something has to look for the `.env` file, and you're right!

You could write your own code to find the file, parse it, and read them into your Node.js app. Or you could look to npm and find a convenient way to rea the variables into your Node.js app in one fell swoop. You'd likely run across the dotenv package on npm, which is a favorite of mine and what recommend you use. You can install it like this `npm install dotenv` .

You could then require this package in your project and use it's `config` function (config also has an alias of load, in case you see that in the wild) look for the `.env` file, read the variables you defined and make them available to your application.

Follow these steps:

1. create a `package.json` file
2. install the dotenv npm package
3. write the code to read the `.env`
4. run the code

It's time to read the `.env` file with a little bit of code. Replace the contents of your `server.js` file with the following code.

```
// server.js
console.log(`Your port is ${process.env.PORT}`); // undefined
const dotenv = require('dotenv');
dotenv.config();
console.log(`Your port is ${process.env.PORT}`); // 8626
```

The code displays the initial value of the `PORT` environment variable, which will be undefined. Then it requires the dotenv package and execute its `config` function, which reads the `.env` file and sets the environment variables. The final line of code displays the `PORT` as 8626.

Error Handling in Express.js

Error handling often doesn't get the attention and prioritization it deserves. Especially for newbie developers, there is more focus on setting up routin route handlers, business logic, optimizing performance, etc. As a result, the equally (if not more) crucial error-handling part will likely be overlooke Striving for the most optimized code and squeezing out every last ounce of performance is all well and good; yet, it's important to remember all it takes one unhandled error leak into your user interface to override all the seconds you helped your users save.

Default Error Handling in Express.js

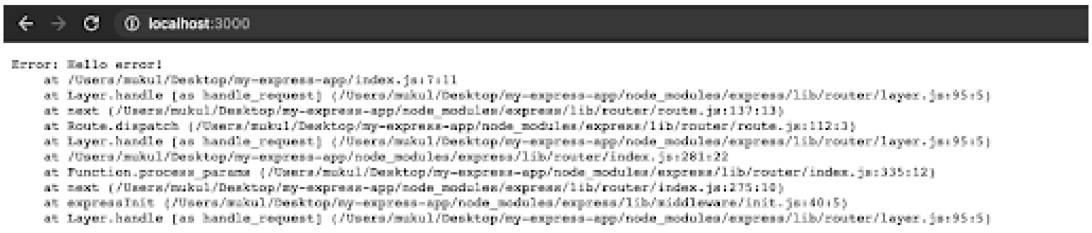
Express implicitly takes care of catching your errors to prevent your application from crashing when it comes to error handling. This is especially true f synchronous route handler code. Let's see how:

Synchronous Code:

Synchronous code refers to statements of code that execute sequentially and one at a time. When an error encounters synchronous code, Expres catches it automatically. Here's an example of a route handler function where we simulate an error condition by throwing an error:

```
app.get('/', (req, res) => {
  throw new Error("Hello error!")
})
```

Express catches this error for us and responds to the client with the error's status code, message, and even the stack trace (for non-productic environments).



```
Error: Hello error!
    at /Users/mukul/Desktop/my-express-app/index.js:7:11
    at Layer.handle [as handle_request] (/Users/mukul/Desktop/my-express-app/node_modules/express/lib/router/layer.js:95:5)
    at next (/Users/mukul/Desktop/my-express-app/node_modules/express/lib/router/route.js:137:13)
    at Route.dispatch (/Users/mukul/Desktop/my-express-app/node_modules/express/lib/router/route.js:112:3)
    at Layer.handle [as handle_request] (/Users/mukul/Desktop/my-express-app/node_modules/express/lib/router/layer.js:95:5)
    at /Users/mukul/Desktop/my-express-app/node_modules/express/lib/router/index.js:281:22
    at Function.process_params (/Users/mukul/Desktop/my-express-app/node_modules/express/lib/router/index.js:335:12)
    at next (/Users/mukul/Desktop/my-express-app/node_modules/express/lib/router/index.js:275:10)
    at expressInit (/Users/mukul/Desktop/my-express-app/node_modules/express/lib/middleware/init.js:40:5)
    at Layer.handle [as handle_request] (/Users/mukul/Desktop/my-express-app/node_modules/express/lib/router/layer.js:95:5)
```

All of this is taken care of thanks to Express's **default built-in error handler middleware** function inserted at the end of your code's middleware stac This automatic handling saves you from bulky try/catch blocks and explicit calls to the in-built middleware (shown below) while also providing som fundamental default error handling functionality.

```
app.get('/', (req, res, next) => {
  try {
      throw new Error("Hello error!")
  }
  catch (error) {
      next(error)
  }
})
```

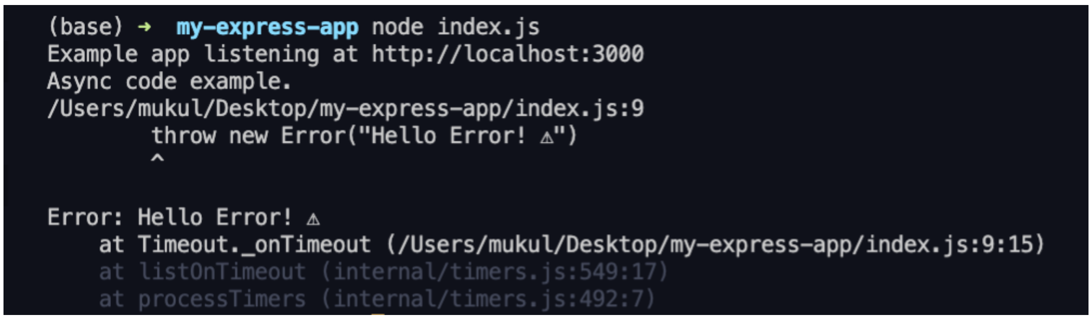
You can also choose to create your own middleware function to specify your error handling logic. We will see that further.

Asynchronous Code:

When writing server-side code, most of your route handlers are likely using asynchronous Javascript logic to read and write files on the server, que databases, and make external API requests. Let's see whether Express can catch errors raised from asynchronous code as well. We'll throw an err from inside the asynchronous *setTimeout()* function and see what happens:

```
app.get('/', (req, res) => {
  setTimeout(() => {
      console.log("Async code example.")
      throw new Error("Hello Error!")
  }, 1000)
})
```

As you can see, our server crashed because Express didn't handle the error for us.

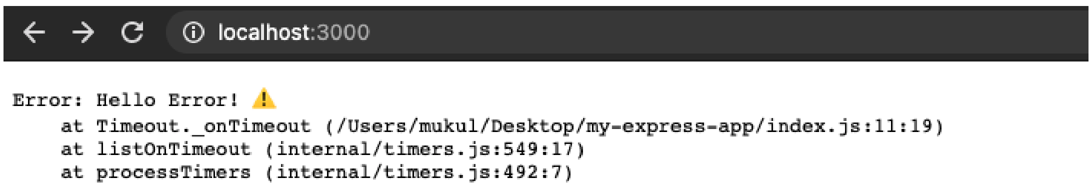


```
(base) ➜  my-express-app node index.js
Example app listening at http://localhost:3000
Async code example.
/Users/mukul/Desktop/my-express-app/index.js:9
        throw new Error("Hello Error! ⚠")
        ^

Error: Hello Error! ⚠
    at Timeout._onTimeout (/Users/mukul/Desktop/my-express-app/index.js:9:15)
    at listOnTimeout (internal/timers.js:549:17)
    at processTimers (internal/timers.js:492:7)
```

For handling errors raised during asynchronous code execution in Express (versions < 5.x), developers need to themselves catch their errors and invol the in-built error handler middleware using the next() function. Here's how:

```
app.get('/', (req, res, next) => {
  setTimeout(() => {
      try {
          console.log("Async code example.")
          throw new Error("Hello Error!")
      } catch (error) { // manually catching
          next(error) // passing to default middleware error handler
      }
  }, 1000)
})
```



```
Error: Hello Error! ⚠
    at Timeout._onTimeout (/Users/mukul/Desktop/my-express-app/index.js:11:19)
    at listOnTimeout (internal/timers.js:549:17)
    at processTimers (internal/timers.js:492:7)
```

This is much better – we caught the error, and our server didn't crash. This does look a little bulky because we used the setTimeout() function demonstrate async behavior. This function does not return a promise and, therefore, can't be chained with a quick *.catch()* function. However, mo libraries that help with async operations return promises these days (e.g., the file system API). Below is an example of a more convenient and commo way of catching errors from promises:

```
const fsPromises = require('fs').promises
app.get('/', (req, res, next) => {
  fsPromises.readFile('./no-such-file.txt')

     .then(data => res.send(data))

     .catch(err => next(err))
})
```

> ***Note:*** *Express 5.0 (currently in alpha) can automatically catch errors (and rejections) thrown by returned Promises.*

Handling Custom Errors

Express's default error-handling middleware is super helpful for beginners to take care of unexpected, unhandled errors. However, different develope and organizations would want their errors handled in their own way – some might want to write these to log files, others might want to alert the user redirect them to another page, or all of the above.

Custom Handling for Each Route:

An obvious, naive way of going about this would be to define your custom error handling logic for each route handler as so:

```
const express = require('express')
const fsPromises = require('fs').promises;

const app = express()
const port = 3000

app.get('/one', (req, res) => {
  fsPromises.readFile('./one.txt')
    .then(data => res.send(data))
    .catch(err => { // error handling logic 1
        console.error(err) // logging error
        res.status(500).send(err)
    })
})

app.get('/two', (req, res) => {
  fsPromises.readFile('./two.txt')
    .then(data => res.send(data))
    .catch(err => { // error handling logic 2
        console.error(err)
        res.redirect('/error') // redirecting user
    })
})

app.get('/error', (req, res) => {
  res.send("Custom error landing page.")
})

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

Here, we specified two different handling logics – one for each route that attempts to read arbitrary files on the server. As you can imagine, this would g too redundant quickly and wouldn't scale well as you add more and more routes.

Adding a Middleware Function for Error Handling

Express comes with a default error handler that takes care of any errors that might be encountered in the application. The default error handler is adde as a middleware function at the end of the middleware function stack.

We can change this default error handling behavior by adding a custom error handler which is a middleware function that takes an error parameter addition to the parameters: `request` , `response` , and the `next()` function. The error handling middleware functions are attached after the rou definitions.

The basic signature of an error-handling middleware function in Express looks like this:

```
function customErrorHandler(error, request, response, next) {

  // Error handling middleware functionality

}
```

When we want to call an error-handling middleware, we pass on the error object by calling the `next()` function with the `error` argument like this:

```
const errorLogger = (error, request, response, next) => {
    console.log( `error ${err.message}`)
    next(error) // calling next middleware
}
```

Let us define three middleware error handling functions and add them to our routes. We have also added a new route that will throw an error as show below:(continued- refer to example in express middleware section)

```
// Error handling Middleware functions
const errorLogger = (error, request, response, next) => {
  console.log( `error ${error.message}`)
  next(error) // calling next middleware
}

const errorResponder = (error, request, response, next) => {
  response.header("Content-Type", 'application/json')

  const status = error.status || 400
  response.status(status).send(error.message)
}

const invalidPathHandler = (request, response, next) => {
  response.status(400)
  response.send('invalid path')
}

app.get('product', (request, response)=>{
  response.sendFile("productsample.html")
})

// handle get request for path /
app.get('/', (request, response) => {
  response.send('response for GET request');
})

app.post('/products', requireJsonContent, (request, response) => {
  ...
})
```

```
app.get('/productswitherror', (request, response) => {
  let error = new Error(`processing error in request at ${request.url}`)
  error.statusCode = 400
  throw error
})

app.use(errorLogger)
app.use(errorResponder)
app.use(invalidPathHandler)
app.listen(PORT, () => {
  console.log(`Server listening at http://localhost:${PORT}`)
})
```

These middleware error handling functions perform different tasks: `errorLogger` logs the error message, `errorResponder` sends the error respons to the client, and `invalidPathHandler` responds with a message for `invalid path` when a non-existing route is requested.

We have next attached these three middleware functions for handling errors to the `app` object by calling the `use()` method after the route definition

To test how our application handles errors with the help of these error handling functions, let us invoke the route wi URL: `localhost:3000/productswitherror` .

Now instead of the default error handler, the first two error handlers get triggered. The first one logs the error message to the console and the secor one sends the error message in the response.

When we request a non-existent route, the third error handler is invoked giving us an error message: `invalid path` .

Conclusion

In this module we discussed one of the main topics in express i.e. middlewares. Then we saw different ways of error handling in express and also how manage environment variables.

Interview Questions

> What is Middleware in express.js?

Middleware is a function that is invoked by the express routing layer before the final request processed.

Middleware functions can perform the following tasks:

1. Execute any code - validation, setting headers, etc.
2. You can make changes to the request (req) and response (res) objects.
3. You can also end the request-response cycle, if rquired.
4. You can call the next middleware function in the stack to proceed and process the final request.

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwis the request will be left hanging.

Types of Middleware:

1. Application-level middleware
2. Router-level middleware
3. Error-handling middleware
4. Built-in middleware
5. Third-party middleware

> What is global Error handling middleware in Express.js?

```
var express = require('express'),
app = express();

app.use(function (err, req, res, next) {
  console.error(err.stack) // error first callback
```

```
    res.status(500).send('Something went wrong!')
  })
```

Thank you !