- ⏱️ 10 minutes
- ✅ Completed

Asynchronous Route Handlers in Express

Up to this point, your Express route handler functions have been synchronous—each statement predictably executes in the order that they're written in. Most Express applications, at some point, need to interact with a database or an API (or both). Interacting with those external resources requires you to write asynchronous code, which in turn requires your route handler functions to be asynchronous.

In modern JavaScript applications, writing asynchronous code means working with Promises and optionally the `async`/`await` keywords. When working with Promises in Express route handlers or middleware functions, special attention needs to be spent on how errors are handled.

When you finish this article, you should be able to:

- Recall that Express cannot process unhandled Promise rejections from within route handler (or middleware) functions;
- Use a Promise `catch` block or a `try`/`catch` statement with `async`/`await` to properly handle errors thrown from within an asynchronous route handler (or middleware) function; and
- Write a wrapper function to simplify catching errors thrown within asynchronous route handler (or middleware) functions.

Calling asynchronous functions or methods within route handlers

To see how to properly call asynchronous functions or methods within route handlers, you need an asynchronous function or method to call.

Creating a simple asynchronous function

In a future article, you'll see how to integrate your Express application with a database. For now keep things as simple as possible by creating a standalone function that wraps the built-in `setTimeout()` function in a Promise:

```
/**
* Asynchronous function that delays for the provided length of
time.
* If the length of time to wait is less than '0', then the
returned
* Promise will reject, otherwise it'll resolve.
* @param {number} timeToWait - The length of time to wait in
milliseconds.
*/
const delay = (timeToWait) => new Promise((resolve, reject) =>
{
 setTimeout(() => {
   if (timeToWait < 0) {
     reject(new Error('An error has occurred!'));
   } else {
     resolve(`All done waiting for ${timeToWait}ms!`);
   }
 }, Math.abs(timeToWait));
});
```

The above `delay()` function accepts a `timeToWait` value and returns a new Promise that calls the `setTimeout()` function passing in the `timeToWait` absolute value. When the `setTimeout()` function call completes, the Promise is resolved if the `timeToWait` value is a positive number or it's rejected if the `timeToWait` value is a negative number.

Note: *The `Math.abs()` function is used to get the absolute value of the `timeToWait` parameter value. Getting the absolute value ensures that the value passed to the `setTimeout()` function is always a positive number. For more information about absolute values, see this Wikipedia page.*

Setting up the Express application

With your `delay()` function in hand, use it to create a simple Express application:

```
// app.js

const express = require('express');

/**
* Asynchronous function that delays for the provided length of
time.
* If the length of time to wait is less than '0', then the
returned
* Promise will reject, otherwise it'll resolve.
```

```
* @param {number} timeToWait - The length of time to wait in
milliseconds.
*/
const delay = (timeToWait) => new Promise((resolve, reject) =>
{
 setTimeout(() => {
   if (timeToWait < 0) {
     reject(new Error('An error has occurred!'));
   } else {
     resolve(`All done waiting for ${timeToWait}ms!`);
   }
 }, Math.abs(timeToWait));
});

// Create the Express app.
const app = express();

// Define routes.

app.get('*', (req, res) => {
  // TODO
});

// Define a port and start listening for connections.

const port = 8080;

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

Note: *If you're following along, don't forget to use npm to install Express (i.e.* `npm install express`*)!*

Now call the `delay()` function within your route handler function. Remember that the `delay()` function returns a Promise, so you can use the Promise `then()` method to execute code when the `delay()` method completes. Within the callback that you pass to the `then()` method, send the value returned from the `delay()` function to the client using the `res.send()` method:

```
app.get('*', (req, res) => {
 delay(5000).then((value) => res.send(value));
});
```

If you start your application (i.e. `node app.js`) and browse to `http://localhost:8080/` you'll see that the request hangs for 5 seconds before the server sends the response "All done waiting for 5000ms!"

Feel free to experiment by varying the number of milliseconds that you're passing to the `delay()` function. For now, continue to pass a positive number; we'll see in just a moment what happens when we pass a negative number.

Using `async`/`await`

Instead of using the Promise `then()` method to execute code when an asynchronous function or method call has completed, you can use the `await` keyword.

Start with adding the `async` keyword to your route handler function to indicate that it's going to make an asynchronous function or method call. Then use the `await` keyword to wait for a result to be returned from the `delay()` function call:

```
app.get('*', async (req, res) => {
 const result = await delay(5000);
 res.send(result);
});
```

If you test your application again you'll see that it behaves the same as it did before—the request hangs for 5 seconds before the server sends the response "All done waiting for 5000ms!"

Catching errors thrown by asynchronous functions or methods

So far, you've stayed on the "happy" path by passing a positive number to the `delay()` function. If you pass a negative number to the `delay()` function, it'll throw an error:

```
app.get('*', async (req, res) => {
 const result = await delay(-5000);
 res.send(result);
});
```

This time when testing your application, the browser will indefinitely hang as it waits for the server to return a response. If you look in the terminal, you'll see that an error occurred:

```
(node:89455) UnhandledPromiseRejectionWarning: Error: An error has
occurred!
    at Timeout._onTimeout ([path to the project
folder]/app.js:13:14)
    at listOnTimeout (internal/timers.js:537:17)
    at processTimers (internal/timers.js:481:7)
(node:89455) UnhandledPromiseRejectionWarning: Unhandled promise
rejection. This error originated either by throwing inside of an
async function without a catch block, or by rejecting a promise
which was not handled with .catch(). To terminate the node process
on unhandled promise rejection, use the CLI flag `--unhandled-
rejections=strict` (see
https://nodejs.org/api/cli.html#cli_unhandled_rejections_mode).
(rejection id: 1)
(node:89455) [DEP0018] DeprecationWarning: Unhandled promise
rejections are deprecated. In the future, promise rejections that
are not handled will terminate the Node.js process with a non-zero
exit code.
```

Node.js is warning you that an unhandled Promise rejection occurred. Furthermore, it's warning that in a future version of Node, unhandled Promise rejections will terminate the Node process (which would result in your application being stopped)!

Luckily, this issue is easy to deal with—simply add a `try/catch` statement around your asynchronous function or method call:

```
app.get('*', async (req, res, next) => {
 try {
   const result = await delay(-5000);
   res.send(result);
 } catch (err) {
   next(err);
 }
});
```

Notice that you need to add the `next` parameter to your route handler function parameter list and pass the caught error to the `next()` method in the `catch` block. Passing the error to the `next()` method allows the Express default error handler process the error.

Note: *When writing custom middleware functions, calling the `next()` method* ***without an argument*** *passes control to the next middleware function. Calling the `next()` method* ***with an argument*** *results in Express handling the current request as an error and skipping any remaining routing and middleware functions.*

The Express default error handler is a special type of middleware function that's responsible for handling errors. In a development environment, the default error handler logs the error to the console and sends a response with an HTTP status code of `500 Internal Server Error` to the client containing the error message along with the stack trace.

If you test your application again, you'll see the default error handler in action as it provides details about the unhandled error that occurred after the `delay()` function has waited for the number of milliseconds that you passed in:

```
Error: An error has occurred!
   at Timeout._onTimeout ([path to the project folder]/app.js:13:14)
   at listOnTimeout (internal/timers.js:537:17)
   at processTimers (internal/timers.js:481:7)
```

If you're not using the `async/await` keywords, you need to call the `catch()` method on the Promise returned by the `delay()` function to handle any thrown errors:

```
app.get('*', (req, res, next) => {
 delay(-5000)
   .then((value) => res.send(value))
   .catch((err) => next(err));
});
```

Again, notice that you need to add the `next` parameter to your route handler function parameter list and call the `next()` method passing in the error caught by the `catch()` method.

If you're only passing the error to the `next()` method, you can simplify your code by passing the reference to the `next()` method directly to the `catch()` method call:

```
app.get('*', (req, res, next) => {
 delay(-5000)
   .then((value) => res.send(value))
```

```
  .catch(next);
});
```
Express is able to automatically catch errors thrown by synchronous route handlers. When performing asynchronous operations within route handlers, it's important to remember that **Express is unable to catch errors thrown by asynchronous route handlers**. Given that, asynchronous route handlers need to catch their own errors and pass them to the `next()` method.
Note: *While all of the examples that you've seen in this article are built around route handlers, everything that's been shown and discussed equally applies to asynchronous custom middleware functions.*

Reducing boilerplate code

Adding a `try/catch` statement to each route handler function that needs to call an asynchronous function or method can result in a lot of boilerplate code. If your application only has a handful of routes that's probably not an issue, but if your application has dozens of routes (or more), it's worth taking a look at how you can reduce the amount of boilerplate code you need to write.

Writing an asynchronous route handler wrapper function

One approach to avoiding writing boilerplate code is to write a simple asynchronous route handler wrapper function to catch errors.

Start by defining a function named `asyncHandler` that accepts a reference to a route handler function and returns a function that defines three parameters, `req`, `res`, and `next`:
```
const asyncHandler = (handler) => {
  return (req, res, next) => {
    // TODO
  };
};
```
Then, within the function that's being returned, call the passed in route handler function (i.e. the `handler` parameter), passing in the `req`, `res`, and `next` parameters:
```
const asyncHandler = (handler) => {
 return (req, res, next) => {
   return handler(req, res, next);
 };
};
```

And finally, call the `catch()` method on the Promise returned from the route handler function passing in the `next` parameter:

```
const asyncHandler = (handler) => {
 return (req, res, next) => {
   return handler(req, res, next).catch(next);
 };
};
```

Remember, passing the `next` parameter to the `catch()` method allows the Express default error handler to process any errors thrown by the route handler function.
Because each of the arrow functions return a single statement,
the `asyncHandler()` function can optionally be written a little more concisely:

```
const asyncHandler = (handler) => (req, res, next) =>
handler(req, res, next).catch(next);
```

Note: *Developers sometimes find the more concise version to be more difficult to read and understand, so if you're working on a team, talk with your teammates and determine which approach is the team's preferred approach.*

Using the asynchronous route handler wrapper function

As a reminder, this is what your asynchronous route handler currently looks like:

```
app.get('*', async (req, res, next) => {
 try {
   const result = await delay(-5000);
   res.send(result);
 } catch (err) {
   next(err);
 }
});
```

Wrapping your asynchronous route handler with your `asyncHandler()` helper function looks like this:

```
app.get('*', asyncHandler(async (req, res) => {
 const result = await delay(5000);
 res.send(result);
}));
```

Because the `asyncHandler()` function is calling the `catch()` method on the Promise that's returned from the asynchronous route handler you can safely remove the `try/catch` statement. This makes asynchronous route handlers cleaner and easier to read and maintain.
Note: *You might wonder how the `asyncHandler()` function can successfully call the `catch()` method after invoking the asynchronous route handler function if the route handler function doesn't explicitly return a Promise. Remember that marking a function or method with the `async` keyword results in that function or method implicitly returning a*

Promise. Your asynchronous route handler is marked as an `async` function, so it implicitly returns a Promise.

What you learned

In this article, you learned

- that Express cannot process unhandled Promise rejections from within a route handler (or middleware) function;
- how to use a Promise `catch` block or a `try`/`catch` statement with `async`/`await` to properly handle errors thrown from within asynchronous route handler (or middleware) function; and
- how to write a wrapper function to simplify catching errors thrown within asynchronous route handler (or middleware) functions.

See also…

While writing a wrapper function for your asynchronous route handler function is easy to do, you can also use an npm package to accomplish the same thing without having to write any extra code. If you're interested to see what this looks like, check out the express-promise-router npm package.