- ⏱️ 13 minutes

Handling Errors in Express

No matter how hard we try, we all make mistakes when writing code. If you're lucky, the coding mistake will break the execution of the application in a very obvious way—crashing the application when testing in the local development environment. If you're unlucky, the coding mistake will go unnoticed, only to surface as an unexpected error when the application is being used by end users.

When an unexpected error occurs, the default error handler in Express will send a response to the browser containing the error message along with the stack trace (if you're not running in a production environment). While the default error handler might work fine in your local development environment, for most applications you'll want to create a custom error handler to precisely control how errors are handled in other environments (i.e. test, staging, or production).

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

- Describe how an error handler function differs from middleware and route handler functions;
- Define a global error-handling function to catch and process unhandled errors; and
- Define a route to handle requests for unknown routes by throwing a 404 NOT FOUND error.

Setting up an example Express application

Let's create a simple application to assist with exploring how to handle errors in Express.

Create a folder for your project (if you haven't already), open a terminal and browse to your project folder, and run the following commands:

```
npm init -y
npm install express@^4.0.0 pug@^2.0.0
npm install nodemon --save-dev
```

Important: *If you're using Git, don't forget to add a* `.gitignore` *file in the root of your project folder that contains an entry to ignore the* `node_modules` *folder!*

The `node_modules` folder tends to be very large and would bloat the Git repository if it was committed and pushed. Ignoring the `node_modules` folder is possible because it can be generated on demand by running the `npm install` command.

Add an `app.js` file to the root of your project containing the following code:

```
// app.js

const express = require('express');

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

// Set the pug view engine.
app.set('view engine', 'pug');

// Define routes.

app.get('/', (req, res) => {
  res.render('index', { title: 'Home' });
});

app.get('/throw-error', (req, res) => {
  throw new Error('An error occurred!');
});

// Define a port and start listening for connections.

const port = 8080;

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

Next, define an npm `start` script in your `package.json` file that uses Nodemon to run the application:

```
{
 "name": "handling-errors-in-express",
 "version": "1.0.0",
 "description": "",
 "main": "app.js",
 "scripts": {
   "start": "nodemon app.js"
 },
 "keywords": [],
 "author": "",
 "license": "ISC",
 "dependencies": {
   "express": "^4.17.1",
   "pug": "^2.0.4"
 },
 "devDependencies": {
   "nodemon": "^2.0.2"
```

```
 }
}
```

Pro Tip: *Creating scripts in the* `package.json` *file allows you to customize and rename often used terminal commands. For example, the above* `"start": "nodemon app.js"` *script allows you to run* `npm start` *instead of* `npx nodemon app.js` *in your terminal to run the application.*

The last bit of set up business is to create a couple of Pug views. Add a `views` folder to the root of the project. Then add two files to the `views` folder: `layout.pug` and `index.pug`:

```
//- layout.pug

doctype html
html
 head
   title Custom Error Handlers - #{title}
 body
   h1 Custom Error Handlers
   h2= title
   div
     block content
//- index.pug

extends layout.pug

block content
 p Welcome to the Custom Error Handlers project!
```

Now you're ready to run and test your application! From the terminal, run the command `npm start`, then browse to the URL `http://localhost:8080/`. You should see the application's Home page.

The default error handler in Express

After an error is thrown or the `next()` method is called with an argument from within a route handler, Express will handle the error using its default error handler.

You can see the default error handler in action by starting the example application (i.e. `npm start`) and browsing to `http://localhost:8080/throw-error`. The default error handler will send a response to the browser containing the error message along with the stack trace:

```
Error: An error occurred!
   at throwError ([path to the project folder]/app.js:14:9)
   at [path to the project folder]/app.js:29:3
   at Layer.handle [as handle_request] ([path to the project
folder]/node_modules/express/lib/router/layer.js:95:5)
```

```
   at next ([path to the project
folder]/node_modules/express/lib/router/route.js:137:13)
   at Route.dispatch ([path to the project
folder]/node_modules/express/lib/router/route.js:112:3)
   at Layer.handle [as handle_request] ([path to the project
folder]/node_modules/express/lib/router/layer.js:95:5)
   at [path to the project
folder]/node_modules/express/lib/router/index.js:281:22
   at Function.process_params ([path to the project
folder]/node_modules/express/lib/router/index.js:335:12)
   at next ([path to the project
folder]/node_modules/express/lib/router/index.js:275:10)
   at expressInit ([path to the project
folder]/node_modules/express/lib/middleware/init.js:40:5)
```

Note: *If you're following along, the placeholder text "path to the project folder" in the above error stack trace information will display the actual absolute path to your project folder.*

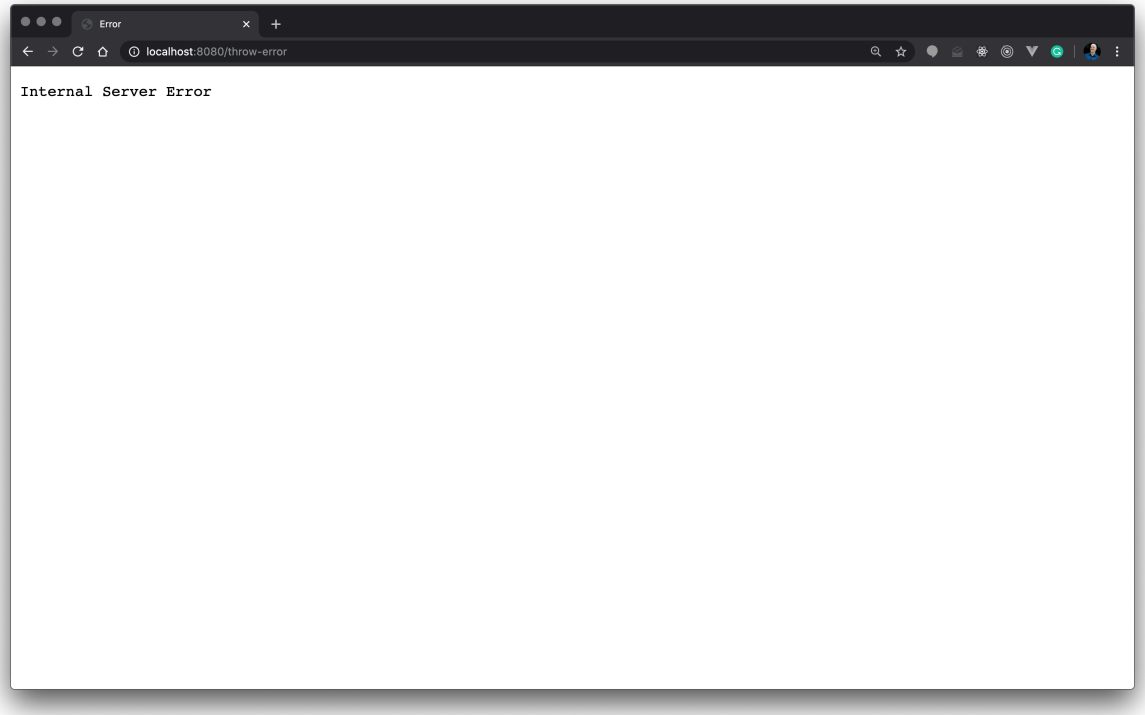
The response will also have an HTTP status code of `500 Internal Server Error`. You can view the response's HTTP status code by inspecting the network information using your browser's developer tools.

Do you need a custom error handler?

If the `NODE_ENV` environment variable is set to "production", then the default error handler will simply return a response with an HTTP status code of `500 Internal Server Error` containing the text "Internal Server Error".

You can see this in action by setting the `NODE_ENV` environment variable before starting the example application:

```
NODE_ENV=production node app.js
```



If an end user were to see the above error message in production, it would likely leave them frustrated and confused. While a custom error handler won't be able to magically resolve unexpected errors for your users, it will allow you to display a friendlier message using your website's layout template (you'll see how to do this later in this article).

Defining a custom error handler will also allow you to log unexpected errors so that you (or someone on your team) can review them periodically to determine if an undetected bug has made its way into production.

Defining a custom error handler

As you've seen in earlier lessons, Express middleware functions define three parameters (`req`, `res`, `next`) and route handlers define two or three parameters (`req`, `res`, and optionally the `next` parameter):

```
// Middleware function.
app.use((req, res, next) => {
  console.log('Hello from a middleware function!');
  next();
});

// Route handler function.
app.get('/', (req, res) => {
```

```
  res.send('Hello from a route handler function!');
});
```

Error handling functions look the same as middleware functions except they define four parameters instead of three—`err`, `req`, `res`, and `next`:

```
app.use((err, req, res, next) => {
  console.error(err);
  res.send('An error occurred!');
});
```

Custom error handler functions have to define four parameters otherwise Express won't recognize the function as an error handler. Route handler function definitions can omit the `next` parameter if it's not going to be used; error handler functions have to include the `next` parameter.

Define error handler functions after all other calls to `app.use()` and all of your application's route definitions:

```
// app.js

const express = require('express');

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

// Set the pug view engine.
app.set('view engine', 'pug');

// Define routes.

app.get('/', (req, res) => {
  res.render('index', { title: 'Home' });
});

app.get('/throw-error', (req, res) => {
  throw new Error('An error occurred!');
});

// Custom error handler.
app.use((err, req, res, next) => {
  console.error(err);
  res.send('An error occurred!');
});

// Define a port and start listening for connections.

const port = 8080;

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

This ensures that your custom error handler will get called to handle errors from any of your application's middleware or route handler functions.

If you test your custom error handler by browsing to `http://localhost:8080/throw-error` you'll see that it sends a response containing the text "An error occurred!".
If you use your browser's developer tools to inspect the response of `http://localhost:8080/throw-error`, you'll notice that the response HTTP status code is `200 OK`, which is the default status code used by Express when sending responses. You can use the `res.status()` method to set a different status code:

```
// Custom error handler.
app.use((err, req, res, next) => {
  console.error(err);
  res.status(err.status || 500);
  res.send('An error occurred!');
});
```

Notice how the `err.status` property is checked to see if it has a value before the status is set to the literal numeric value `500`. Giving priority to the `err.status` property allows code elsewhere in the application to throw an error that includes the specific HTTP status code to send to the client.

You can return an HTML response instead of plain text by rendering a Pug view:

```
// Custom error handler.
app.use((err, req, res, next) => {
 console.error(err);
 res.status(err.status || 500);
 const isProduction = process.env.NODE_ENV === 'production';
 res.render('error', {
   title: 'Server Error',
   message: isProduction ? null : err.message,
   error: isProduction ? null : err,
 });
});
```

Be sure to add the `error.pug` view to the `views` folder:

```
//- error.pug

extends layout.pug

block content
 div
   p= message || 'An unexpected error occurred on the server.'
 if stack
   h3 Stack Trace
   pre= stack
```

Notice that the error `message` and `stack` properties are only being passed to the view if the `NODE_ENV` environment variable isn't set to "production". For security reasons, it's important to avoid leaking potentially sensitive information about your application.
If you test your custom error handler again by browsing to `http://localhost:8080/throw-error` you'll see that it sends an HTML response containing information about the error that was thrown.

To test how the error handler will behave in the production environment, set the `NODE_ENV` environment variable to "production" before starting the example application:

```
NODE_ENV=production node app.js
```

Defining multiple custom error handlers

Express allows you to define more than one custom error handler which is useful if you need to handle specific types of errors differently. It's also useful for creating an error handler to perform a specific error handling task. Let's look at an example of defining a second error handler that's responsible for logging errors.

Error handlers, like route handlers, are executed by Express in the order that they're defined in, so defining a new error handler before the existing handler ensures that it'll be called first:

```
// Custom error handlers.

// Error handler to log errors.
app.use((err, req, res, next) => {
  if (process.env.NODE_ENV === 'production') {
    // TODO Log the error to the database.
  } else {
    console.error(err);
  }
  next(err);
});

// Generic error handler.
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  const isProduction = process.env.NODE_ENV === 'production';
  res.render('error', {
    title: 'Server Error',
    message: isProduction ? null : err.message,
    stack: isProduction ? null : err.stack,
  });
});
```

The new error handler simply uses the `console.error()` method to log errors to the console, provided that the `NODE_ENV` environment variable isn't set to "production". In the production environment, there's a TODO comment to log the error to the database. The `console.error()` method call in the existing error handler was removed; logging errors is now the responsibility of the new error handler.

Note: *Logging errors to a database—or another type of data store—is a common practice in production environments. Doing this allows a developer or system administrator to*

periodically review a log of application errors to determine if there are any issues that might need to be looked at in more detail. There are many ways to handle error logging, ranging from npm logging packages (e.g. winston) to full-blown application monitoring cloud-based services.
Also notice that the new error handler calls the `next()` method passing in the `err` parameter (the current error) which passes control to the next error handler. An error handler needs to call `next()` or return a response. Failing to do this will result in the request "hanging" and consuming resources on the server.

Handling "Page Not Found" errors

A common feature for applications to implement is to present a friendly "Page Not Found" message to end users when a request can't be matched to one of the application's defined routes. Let's see how to implement this feature using a combination of a middleware function and an error handler function.

First, add a new middleware function after the last route in your application (but before any of your error handlers):

```
// Catch unhandled requests and forward to error handler.
app.use((req, res, next) => {
 const err = new Error('The requested page couldn\'t be
found.');
 err.status = 404;
 next(err);
});
```

Placing this middleware function after all of your routes means that this middleware function will only be invoked if a request fails to match any of your routes.

Notice that the middleware function creates a new Error object and sets a `status` property on the object to the literal number `404`. `404` is the HTTP status code for "Not Found" responses indicating that the requested resource could not be found. After setting the `status` property, the `next()` method is called with the `err` variable passed as an argument. Remember that 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.
At this point, you can test your "Page Not Found" middleware function by browsing to `http://localhost:8080/some-unknown-page` (or really any path that doesn't match one of your application's configured routes). You should see your "Server Error" page displaying the message "The requested page couldn't be found."

Creating a "Page Not Found" page

While the current solution works, a more elegant solution would be to present a specific "Page Not Found" page to the end user.

To do this, define another error handler—in between the logging and generic error handlers—for handling 404 errors:

```
// Error handler for 404 errors.
app.use((err, req, res, next) => {
  if (err.status === 404) {
    res.status(404);
    res.render('page-not-found', {
      title: 'Page Not Found',
    });
  } else {
    next(err);
  }
});
```

This error handler starts by checking if the `err.status` property is set to `404`—which indicates that the current error is a "Not Found" error. If the current error is a "Not Found" error, then it sets the response HTTP status code to `404` and calls the `res.render()` method to render the `page-not-found` view (you'll create that view in just a bit). Otherwise, the `next()` method is called with the `err` parameter passed as an argument, which passes control to the next error handler.

Before testing your new error handler, don't forget to add a new view named `page-not-found.pug` to the `views` folder with the following content:

```
//- page-not-found.pug

extends layout.pug

block content
 div
   p Sorry, we couldn't find the page that you requested.
```

Now if you test your application again by browsing to `http://localhost:8080/some-unknown-page` (or any path that doesn't match one of your application's configured routes) you should see your new "Page Not Found" page.

For your reference, here's the final version of the `app.js` file:

```
// app.js

const express = require('express');

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

// Set the pug view engine.
app.set('view engine', 'pug');
```

```
// Define routes.

app.get('/', (req, res) => {
  res.render('index', { title: 'Home' });
});

app.get('/throw-error', (req, res) => {
  throw new Error('An error occurred!');
});

// Catch unhandled requests and forward to error handler.
app.use((req, res, next) => {
  const err = new Error('The requested page couldn\'t be
found.');
  err.status = 404;
  next(err);
});

// Custom error handlers.

// Error handler to log errors.
app.use((err, req, res, next) => {
  if (process.env.NODE_ENV === 'production') {
    // TODO Log the error to the database.
  } else {
    console.error(err);
  }
  next(err);
});

// Error handler for 404 errors.
app.use((err, req, res, next) => {
  if (err.status === 404) {
    res.status(404);
    res.render('page-not-found', {
      title: 'Page Not Found',
    });
  } else {
    next(err);
  }
});

// Generic error handler.
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  const isProduction = process.env.NODE_ENV === 'production';
  res.render('error', {
    title: 'Server Error',
```

```
    message: isProduction ? null : err.message,
    stack: isProduction ? null : err.stack,
  });
});

// Define a port and start listening for connections.

const port = 8080;

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

What you learned

In this article, you learned

- how an error handler function differs from middleware and route handler functions;
- how to define a global error-handling function to catch and process unhandled errors; and
- how to define a route to handle requests for unknown routes by throwing a 404 NOT FOUND error.