

[Newsletter](#)[Menu](#)

Using Async/await in Express

22ND FEB 2022

Have you noticed you write a lot of asynchronous code in Express request handlers? This is normal because you need to communicate with the database, the file system, and other APIs.

When you have so much asynchronous code, it helps to use Async/await. It makes your code easier to understand.

Today, I want to share how to use async/await in an Express request handler.

Note: Before you continue, you need to know what Async/await is. If you don't know, you can read [this article](#) for more information.

Update: [Express 5.0 baked the handling of async errors into the framework](#) itself — the error will be sent to the error handling middleware even if the Promise rejects. This means the rest of this article is obsolete once Express 5 comes out of beta.

I'll update this article again Express 5 is out of beta.

Using Async/await with a request handler

To use Async/await, you need to use the `async` keyword when you define a request handler. (Note: These request handlers are

known as called “controllers”. I prefer calling them request handlers because request handlers are more explicit).

```
app.post('/testing', async (req, res) => {  
  // Do something here  
})
```

Once you have the `async` keyword, you can `await` something immediately in your code.

```
app.post('/testing', async (req, res) => {  
  const user = await User.findOne({ email: req.body.email  
})  
})
```

Handling Async errors

Let's say you want to create a user through a POST request. To create a user, you need to pass in a `firstName` and an `email` address. Your Mongoose Schema looks like this:

```
const userSchema = new Schema({  
  email: {  
    type: String,  
    required: true,  
    unique: true  
  },  
  firstName: {  
    type: String,  
    required: true  
  }  
})
```

Here's your request handler:

```
app.post('/signup', async (req, res) => {  
  const { email, firstName } = req.body  
  const user = new User({ email, firstName })  
  const ret = await user.save()  
  res.json(ret)  
})
```

Now, let's say you send a request that lacks an email address to your server.

```
fetch('/signup', {
  method: 'post'
  headers: { 'Content-Type': 'application/json' }
  body: JSON.stringify({
    firstName: 'Zell'
  })
})
```

This request results in an error. Unfortunately, Express will not be able to handle this error. You'll receive a log like this:

```
(node:3502) UnhandledPromiseRejectionWarning: ValidationError: User validation failed: email: Path `email` is required.
    at new ValidationError (/Users/zellwk/Desktop/test/node_modules/mongoose/lib/error/validation.js:30:11)
    at model.Document.invalidate (/Users/zellwk/Desktop/test/node_modules/mongoose/lib/document.js:2292:32)
    at p.doValidate.skipSchemaValidators (/Users/zellwk/Desktop/test/node_modules/mongoose/lib/document.js:2141:17)
    at /Users/zellwk/Desktop/test/node_modules/mongoose/lib/schematype.js:1037:9
    at process._tickCallback (internal/process/next_tick.js:61:11)
(node:3502) 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(). (rejection id: 2)
(node:3502) [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.
```

To handle an error in an asynchronous function, you need to catch the error first. You can do this with `try/catch`.

```
app.post('/signup', async (req, res) => {
  try {
    const { email, firstName } = req.body
    const user = new User({ email, firstName })
    const ret = await user.save()
    res.json(ret)
  } catch (error) {
    console.log(error)
  }
})
```

```
{ ValidationError: User validation failed: email: Path `email` is required.
  at ValidationError.inspect (/Users/zellwk/Desktop/test/node_modules/mongoose/lib/error/validation.js:59:24)
  at formatValue (internal/util/inspect.js:491:31)
  at inspect (internal/util/inspect.js:189:10)
  at Object.formatWithOptions (util.js:84:12)
  at Console.<anonymous function> (console.js:191:15)
  at Console.log (console.js:202:31)
  at app.post (/Users/zellwk/Desktop/test/server.js:67:13)
  at process._tickCallback (internal/process/next_tick.js:68:7)
errors:
  { email:
    { ValidatorError: Path `email` is required.
      at new ValidatorError (/Users/zellwk/Desktop/test/node_modules/mongoose/lib/error/validator.js:29:11)
      at validate (/Users/zellwk/Desktop/test/node_modules/mongoose/lib/schematype.js:1034:13)
      at /Users/zellwk/Desktop/test/node_modules/mongoose/lib/schematype.js:1088:11
      at Array.forEach (<anonymous>)
      at SchemaString.SchemaType.doValidate (/Users/zellwk/Desktop/test/node_modules/mongoose/lib/schematype.js:1043:14)
      at /Users/zellwk/Desktop/test/node_modules/mongoose/lib/document.js:2134:9
      at process._tickCallback (internal/process/next_tick.js:61:11)
      message: 'Path `email` is required.',
      name: 'ValidatorError',
      properties: [Object],
      kind: 'required',
      path: 'email',
      value: undefined,
      reason: undefined,
      [Symbol(mongoose:validatorError)]: true } },
    _message: 'User validation failed',
    name: 'ValidationError' }
```

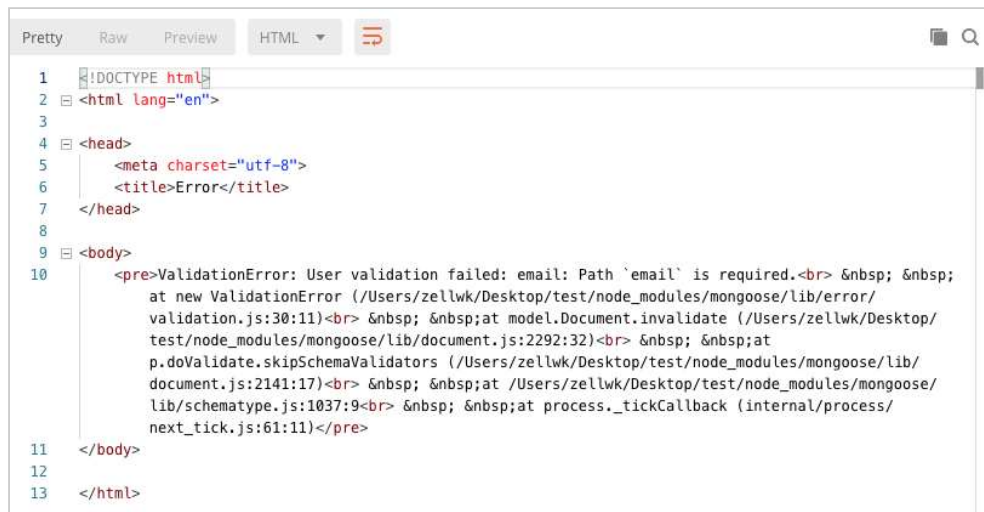
Next, you pass the error into an Express error handler with the `next` argument.

```
app.post('/signup', async (req, res, next) => {
  try {
    const { email, firstName } = req.body
    const user = new User({ email, firstName })
    const ret = await user.save()
    res.json(ret)
  } catch (error) {
    // Passes errors into the error handler
    return next(error)
  }
})
```

If you did not write a custom error handler yet, Express will handle the error for you with its default error handler. (Though I recommend you write a custom error handler. You can learn more about it [here](https://zellwk.com/blog/async-await-express/)).

Express's default error handler will:

1. Set the HTTP status to 500
2. Send a Text response back to the requester
3. Log the text response in the console



```
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="utf-8">
6   <title>Error</title>
7 </head>
8
9 <body>
10  <pre>ValidationError: User validation failed: email: Path `email` is required.<br> &nbsp; &nbsp; &nbsp;
    at new ValidationError (/Users/zellwk/Desktop/test/node_modules/mongoose/lib/error/
    validation.js:30:11)<br> &nbsp; &nbsp; &nbsp;at model.Document.invalidate (/Users/zellwk/Desktop/
    test/node_modules/mongoose/lib/document.js:2292:32)<br> &nbsp; &nbsp; &nbsp;at
    p.doValidate.skipSchemaValidators (/Users/zellwk/Desktop/test/node_modules/mongoose/lib/
    document.js:2141:17)<br> &nbsp; &nbsp; &nbsp;at /Users/zellwk/Desktop/test/node_modules/mongoose/
    lib/schematype.js:1037:9<br> &nbsp; &nbsp; &nbsp;at process._tickCallback (internal/process/
    next_tick.js:61:11)</pre>
11 </body>
12
13 </html>
```

I used Postman to send a request to my server. Here's the text response back from the server.

```
POST /signup 500 10.457 ms - 745
ValidationError: User validation failed: email: Path `email` is required.
  at new ValidationError (/Users/zellwk/Desktop/test/node_modules/mongoose/lib/error/validation.js:30:11)
  at model.Document.invalidate (/Users/zellwk/Desktop/test/node_modules/mongoose/lib/document.js:2292:32)
  at p.doValidate.skipSchemaValidators (/Users/zellwk/Desktop/test/node_modules/mongoose/lib/document.js:2141:17)
  at /Users/zellwk/Desktop/test/node_modules/mongoose/lib/schematype.js:1037:9
  at process._tickCallback (internal/process/next_tick.js:61:11)
```

Notice the 500 HTTP Status log in this image. This tells me Express's default handler changed the HTTP Status to 500. The log is from Morgan. I talked about Morgan in detail [here](#).

Handling two or more async errors

If you need to handle two `await` statements, you might write this code:

```
app.post('/signup', async (req, res, next) => {
  try {
    await firstThing()
  } catch (error) {
    return next(error)
  }
})
```

```
    try {  
      await secondThing()  
    } catch (error) {  
      return next(error)  
    }  
  })
```

This is unnecessary. If `firstThing` results in an error, the request will be sent to an error handler immediately. You would not

trigger a call for `secondThing`. If `secondThing` results in an error, `firstThing` would not have triggered an error.

This means: Only one error will be sent to the error handler. It also means we can wrap all `await` statements in ONE `try/catch` statement.

```
app.post('/signup', async (req, res, next) => {  
  try {  
    await firstThing()  
    await secondThing()  
  } catch (error) {  
    return next(error)  
  }  
})
```

Cleaning up

It sucks to have a `try/catch` statement in each request handler. They make the request handler seem more complicated than it has to be.

A simple way is to change the `try/catch` into a promise. This feels more friendly.

```
app.post('/signup', async (req, res, next) => {  
  async function runAsync () {  
    await firstThing()  
    await secondThing()  
  }  
})
```

```
runAsync().catch(next)
})
```

But it's a chore to write `runAsync` for every Express handler. We can abstract it into a wrapper function. And we can attach this wrapper function to each request handler

```
function runAsyncWrapper (callback) {
  return function (req, res, next) {

    callback(req, res, next)
      .catch(next)
  }
}

app.post('/signup', runAsyncWrapper(async(req, res) => {
  await firstThing()
  await secondThing()
}))
)
```

Express Async Handler

You don't have to write `runAsyncWrapper` code each time you write an express app either. [Alexei Bazhenov](#) has created a package called [express-async-handler](#) that does the job in a slightly more robust way. (It ensures `next` is always the last argument).

Update: I found another package that's much easier to use compared to `express-async-handler`. More on this [below](#).

To use `express-async-handler`, you have to install it first:

```
npm install express-async-handler --save
```

Using it in your app:

```
const asyncHandler = require('express-async-handler')
```

```
app.post(
  '/signup',
  asyncHandler(async (req, res) => {
    await firstThing()
    await secondThing()
  })
)
```

I don't like to write `asyncHandler`. It's quite long. My obvious solution is to abbreviate `asyncHandler` to `ash`.

If you're fancier, you can consider using [@awaitjs/express](#) by [Valeri Karpov](#). It adds methods like `getAsync` and `postAsync` to Express so you don't have to use `express-async-handler`.

Express Async Errors

There's a package called [express-async-errors](#) that makes Express errors much easier to handle. You just have to require it once and it'll take care of the rest.

Installing it:

```
npm install express-async-errors -S
```

Using it:

```
const express = require('express')
require('express-async-errors')
```

That's it!

If you enjoyed this article, please support me by sharing this article [Twitter](#) or [buying me a coffee](#) 😊. If you spot a typo, I'd appreciate if you can [correct it on GitHub](#). Thank you!

Want to become a better Frontend Developer?

Don't worry about where to start. I'll send you a library of articles frontend developers have found useful!

- 60+ CSS articles
- 60+ JavaScript articles

I'll also send you one article every week to help you improve your FED skills crazy fast!

First Name

Email Address

Send me the articles!

[< Serving HTTPS locally with Node](#)

[Why I stopped using Operator Mono >](#)

About Zell

Home
About
Contact

Things I made

Courses
Libraries

Newsletter

Email
RSS

Social Media

<https://zellwk.com/blog/async-await-express/>

[Twitter](#)

[Github](#)

[Youtube](#)

© 2020 [Zell Liew](#) · [Terms](#)