How to handle errors in Express

JUNE 15, 2020 BY ROBIN WIERUCH - EDIT THIS POST

This tutorial is part 2 of 2 in this series.

Part 1: Creating a REST API with Express.js and MongoDB

This tutorial is part 2 of 2 in this series.

Part 1: Creating a REST API with Express.js and PostgreSQL

Handling errors in Express is often an afterthought. However, it should usually start with your Express routes, because if an error happens, your user needs to be notified about it. With the right techniques at hand, error handling in Express becomes pretty straight forward.

There are several scenarios why an error might happen. It can be that a user doesn't provide all information for a RESTful request, that your ORM/ODM layer (e.g. Mongoose) cannot perform the desired operation (due to validation or some other restrictions) on the database, or that a user is not authorized to perform a certain operation on a RESTful resource.

In this section, we will go through a database validation case which we created earlier when we designed our database models, where a user isn't allowed to create a message entity with an empty text. First, try the to execute the following cURL operation on the command line:

```
curl -X POST -H "Content-Type:application/json" http://localhost:3000/messa
```

In this case, everything works without any error, because a `text` is provided. However, if you leave the `text` empty, you will get a different result:

```
curl -X POST -H "Content-Type:application/json" http://localhost:3000/messa
```

The command line may even get stuck, because no response is delivered from the REST API. There may be a message like: "*Empty reply from server*". In the loggings of your server, you may see something like "*UnhandledPromiseRejectionWarning: Unhandled promise rejection.*". If we double check the *src/models/message.js* file, we see that it's not allowed to create a message with an empty `text` property, because it's required:

```
...

const messageSchema = new mongoose.Schema(
  {
    text: {
      type: String,
      required: true,
    },
    user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  },
  { timestamps: true },
);

...
```

Because of this database validation, Mongoose throws an error if there is an empty `text`, which needs to be addressed in our Express route where we create the message in the *src/routes/message.js* file:

```
router.post('/', async (req, res) => {
  let message;
  try {
    message = await req.context.models.Message.create({
```

```
      text: req.body.text,
      user: req.context.me.id,
    });
  } catch (error) {
    return res.status(400).json({ error: error.toString() });
  }

  return res.send(message);
});
```

Try again to execute the previous cURL operation. You may receive a similar output on the command line now:

```
{"error":"ValidationError: text: Path `text` is required."}
```

That's some progress, because we went from an unhandled error, that originated in our database layer and which left us with a stuck server, to handling the error in this particular Express route. Since we are using async/await, it's a common practice to use a try/catch block for these cases. In this case, we just returned a 400 HTTP status code which is commonly used for validation errors which are caused by a wrong input from a client application.

If we would want to have error handling for each of our Express routes, we would end up with lots of try/catch blocks which mostly will perform all the same error handling for us. This wouldn't scale at all for a larger Express application with lots of REST API endpoints. Fortunately, we have Express' middleware to unify this error handling at one place in our *src/index.js* file:

```
...

app.use('/session', routes.session);
app.use('/users', routes.user);
app.use('/messages', routes.message);

app.use((error, req, res, next) => {
  return res.status(500).json({ error: error.toString() });
});

...
```

Express handles any route with four arguments as error handling middleware, so it's important to list all four arguments for the error handling callback function here. It's also important to list this middleware after your REST API routes, because only this way all the errors happening in your REST API endpoints can be delegated to this error handling middleware.

Let's get back to creating a message in *src/routes/message.js*. If you try the cURL request, you will not see the error handling happening if you don't catch any error there. In Express, errors have to be explicitly send via the `next` function to the middleware. Fortunately we don't need to use a try/catch block but just use the promise's catch method instead:

```
router.post('/', async (req, res, next) => {
  const message = await req.context.models.Message.create({
    text: req.body.text,
    user: req.context.me.id,
  }).catch(next);

  return res.send(message);
});
```

Now you should see the returned error when you try to create a message without a `text` again. This is already the gist of error handling in Express, however, I want to show you a few more things.

First of all, we return always a generic HTTP status code 500 here. This may be alright for most validation errors that originate from our database, however, it shouldn't be the default case for all errors. Let's go through this scenario with another status code. Therefore, we will create a new Express route in our *src/index.js* file:

```
app.use('/session', routes.session);
app.use('/users', routes.user);
app.use('/messages', routes.message);

app.get('/some-new-route', function (req, res, next) {
  res.status(301).redirect('/not-found');
});

app.use((error, req, res, next) => {
  return res.status(500).json({ error: error.toString() });
});
```

Visiting this route in your browser will lead to a redirect to a 404 not found page (which we haven't implemented). A 301 HTTP status code always indicates a redirect and Express' `redirect` method lets us perform this redirect programmatically.

Now we want to generalize the redirect for all routes that are not matched by our API. Therefore we can use a wildcard route with an * asterisk, but we need also make sure to use this route as the last route of all our routes:

```
app.use('/session', routes.session);
app.use('/users', routes.user);
app.use('/messages', routes.message);

app.get('*', function (req, res, next) {
  res.status(301).redirect('/not-found');
});
```

Last, we could make our middleware deal with this case and let the wildcard route just throw an error:

```
app.get('*', function (req, res, next) {
  const error = new Error(
    `${req.ip} tried to access ${req.originalUrl}`,
  );

  error.statusCode = 301;

  next(error);
});

app.use((error, req, res, next) => {
  if (!error.statusCode) error.statusCode = 500;

  if (error.statusCode === 301) {
    return res.status(301).redirect('/not-found');
  }

  return res
    .status(error.statusCode)
    .json({ error: error.toString() });
});
```

If no `statusCode` property is available at the error object (which is the default), we will set the status code of the HTTP header to 500. If there is a status code available, we will use this status code for the response. In the special case of having a 301 status code, we will perform a redirecting response.

In the case of our message creation, we may want to specify a 400 HTTP status code for the error, as we did before, and therefore adapt the code in the *src/routes/message.js* file:

```
router.post('/', async (req, res, next) => {
  const message = await req.context.models.Message.create({
    text: req.body.text,
    user: req.context.me.id,
  }).catch((error) => {
    error.statusCode = 400;
    next(error);
  });

  return res.send(message);
});
```

Optionally you could extract this as a reusable custom error which could be located in a *src/utils/errors.js* file. This new error class extends the commonly used JavaScript `Error` class. The only feature we add here is the 400 HTTP status code:

```
export class BadRequestError extends Error {
  constructor(error) {
    super(error.message);

    this.data = { error };
    this.statusCode = 400;
```

```
  }
}
```

Then again in the *src/routes/message.js* file, we could import this new error class and use it for the error handling. This way, we can react to the error and attach an appropriate status code to it:

```
...

import { BadRequestError } from '../utils/errors';

...

router.post('/', async (req, res, next) => {
  const message = await req.context.models.Message.create({
    text: req.body.text,
    user: req.context.me.id,
  }).catch((error) => next(new BadRequestError(error)));

  return res.send(message);
});
```

If we would have to deal with different scenarios here, so not only a validation error but also other errors, we could implement the errors as classes with appropriate HTTP status codes again, and then decide conditionally on which error we want to use based on the thrown error coming from the database. After all, we always have control about which errors we pass to our error handling middleware and the middleware decides what to return to our users.

Exercises:

- Confirm your source code for the last section. Be aware that the project cannot run properly in the Sandbox, because there is no database.
  - Confirm your changes from the last section.
- Extend all other API endpoints where we perform database requests with proper error handling.
- Go through potential HTTP status codes that you may need for your error cases.

Show Comments

KEEP READING ABOUT NODE ›

CUSTOM ERRORS IN JAVASCRIPT

There are two error handling scenarios in JavaScript. Either an error is thrown from a third-party (e.g. library, database, API) or you want to throw an error yourself. While you have the error at...

CREATING A REST API WITH EXPRESS.JS AND POSTGRESQL

Node + Express + PostgreSQL is a powerful tech stack for backend applications to offer CRUD operations. It gives you everything to expose an API (Express routes), to add business logic (Express...