Error Handling

Error Handling refers to how Express catches and processes errors that occur both synchronously and asynchronously. Express comes with a default error handler so you don't need to write your own to get started.

Catching Errors

It's important to ensure that Express catches all errors that occur while running route handlers and middleware.

Errors that occur in synchronous code inside route handlers and middleware require no extra work. If synchronous code throws an error, then Express will catch and process it. For example:

```
app.get('/', (req, res) => {
  throw new Error('BROKEN') // Express will catch this on its
own.
})
```

For errors returned from asynchronous functions invoked by route handlers and middleware, you must pass them to the `next()` function, where Express will catch and process them. For example:

```
app.get('/', (req, res, next) => {
  fs.readFile('/file-does-not-exist', (err, data) => {
    if (err) {
      next(err) // Pass errors to Express.
    } else {
      res.send(data)
    }
  })
})
```

Starting with Express 5, route handlers and middleware that return a Promise will call `next(value)` automatically when they reject or throw an error. For example:

```
app.get('/user/:id', async (req, res, next) => {
  const user = await getUserById(req.params.id)
  res.send(user)
})
```

If `getUserById` throws an error or rejects, `next` will be called with either the thrown error or the rejected value. If no rejected value is provided, `next` will be called with a default Error object provided by the Express router.

If you pass anything to the `next()` function (except the string `'route'`), Express regards the current request as being an error and will skip any remaining non-error handling routing and middleware functions.

If the callback in a sequence provides no data, only errors, you can simplify this code as follows:

```
app.get('/', [
  function (req, res, next) {
    fs.writeFile('/inaccessible-path', 'data', next)
  },
  function (req, res) {
    res.send('OK')
  }
])
```

In the above example, `next` is provided as the callback for `fs.writeFile`, which is called with or without errors. If there is no error, the second handler is executed, otherwise Express catches and processes the error.

You must catch errors that occur in asynchronous code invoked by route handlers or middleware and pass them to Express for processing. For example:

```
app.get('/', (req, res, next) => {
  setTimeout(() => {
```

```
    try {
      throw new Error('BROKEN')
    } catch (err) {
      next(err)
    }
  }, 100)
})
```

The above example uses a `try...catch` block to catch errors in the asynchronous code and pass them to Express. If the `try...catch` block were omitted, Express would not catch the error since it is not part of the synchronous handler code.

Use promises to avoid the overhead of the `try...catch` block or when using functions that return promises. For example:

```
app.get('/', (req, res, next) => {
  Promise.resolve().then(() => {
    throw new Error('BROKEN')
  }).catch(next) // Errors will be passed to Express.
})
```

Since promises automatically catch both synchronous errors and rejected promises, you can simply provide `next` as the final catch handler and Express will catch errors, because the catch handler is given the error as the first argument.

You could also use a chain of handlers to rely on synchronous error catching, by reducing the asynchronous code to something trivial. For example:

```
app.get('/', [
  function (req, res, next) {
    fs.readFile('/maybe-valid-file', 'utf-8', (err, data) => {
      res.locals.data = data
      next(err)
    })
  },
  function (req, res) {
    res.locals.data = res.locals.data.split(',')[1]
```

```
    res.send(res.locals.data)
  }
])
```

The above example has a couple of trivial statements from the `readFile` call. If `readFile` causes an error, then it passes the error to Express, otherwise you quickly return to the world of synchronous error handling in the next handler in the chain. Then, the example above tries to process the data. If this fails, then the synchronous error handler will catch it. If you had done this processing inside the `readFile` callback, then the application might exit and the Express error handlers would not run.

Whichever method you use, if you want Express error handlers to be called in and the application to survive, you must ensure that Express receives the error.

The default error handler

Express comes with a built-in error handler that takes care of any errors that might be encountered in the app. This default error-handling middleware function is added at the end of the middleware function stack.

If you pass an error to `next()` and you do not handle it in a custom error handler, it will be handled by the built-in error handler; the error will be written to the client with the stack trace. The stack trace is not included in the production environment.

> Set the environment variable `NODE_ENV` to `production`, to run the app in production mode.

When an error is written, the following information is added to the response:

- The `res.statusCode` is set from `err.status` (or `err.statusCode`). If this value is outside the 4xx or 5xx range, it will be set to 500.
- The `res.statusMessage` is set according to the status code.
- The body will be the HTML of the status code message when in production environment, otherwise will be `err.stack`.
- Any headers specified in an `err.headers` object.

If you call `next()` with an error after you have started writing the response (for example, if you encounter an error while streaming the response to the client), the Express default error handler closes the connection and fails the request.

So when you add a custom error handler, you must delegate to the default Express error handler, when the headers have already been sent to the client:

```
function errorHandler (err, req, res, next) {
  if (res.headersSent) {
    return next(err)
  }
  res.status(500)
  res.render('error', { error: err })
}
```

Note that the default error handler can get triggered if you call `next()` with an error in your code more than once, even if custom error handling middleware is in place.

Other error handling middleware can be found at Express middleware.

Writing error handlers

Define error-handling middleware functions in the same way as other middleware functions, except error-handling functions have four arguments instead of three: `(err, req, res, next)`. For example:

```
app.use((err, req, res, next) => {
  console.error(err.stack)
  res.status(500).send('Something broke!')
})
```

You define error-handling middleware last, after other `app.use()` and routes calls; for example:

```
const bodyParser = require('body-parser')
const methodOverride = require('method-override')
```

```
app.use(bodyParser.urlencoded({
  extended: true
}))
app.use(bodyParser.json())
app.use(methodOverride())
app.use((err, req, res, next) => {
  // logic
})
```

Responses from within a middleware function can be in any format, such as an HTML error page, a simple message, or a JSON string.

For organizational (and higher-level framework) purposes, you can define several error-handling middleware functions, much as you would with regular middleware functions. For example, to define an error-handler for requests made by using `XHR` and those without:

```
const bodyParser = require('body-parser')
const methodOverride = require('method-override')

app.use(bodyParser.urlencoded({
  extended: true
}))
app.use(bodyParser.json())
app.use(methodOverride())
app.use(logErrors)
app.use(clientErrorHandler)
app.use(errorHandler)
```

In this example, the generic `logErrors` might write request and error information to `stderr`, for example:

```
function logErrors (err, req, res, next) {
  console.error(err.stack)
  next(err)
}
```

Also in this example, `clientErrorHandler` is defined as follows; in this case, the error is explicitly passed along to the next one.

Notice that when ***not*** calling “next” in an error-handling function, you are responsible for writing (and ending) the response. Otherwise, those requests will “hang” and will not be eligible for garbage collection.

```
function clientErrorHandler (err, req, res, next) {
  if (req.xhr) {
    res.status(500).send({ error: 'Something failed!' })
  } else {
    next(err)
  }
}
```

Implement the “catch-all” `errorHandler` function as follows (for example):

```
function errorHandler (err, req, res, next) {
  res.status(500)
  res.render('error', { error: err })
}
```

If you have a route handler with multiple callback functions, you can use the `route` parameter to skip to the next route handler. For example:

```
app.get('/a_route_behind_paywall',
  (req, res, next) => {
    if (!req.user.hasPaid) {
      // continue handling this request
      next('route')
    } else {
      next()
    }
  }, (req, res, next) => {
    PaidContent.find((err, doc) => {
```

```
      if (err) return next(err)
      res.json(doc)
    })
  })
```

In this example, the `getPaidContent` handler will be skipped but any remaining handlers in `app` for `/a_route_behind_paywall` would continue to be executed.