# Handling and Logging Errors

So, in this section, you learned that:

- Our applications don't run in an ideal world. Unexpected errors can happen as a result of bugs in our code or issues in the running environment. For example, our MongoDB server may shut down, or a remote HTTP service we call may go down.

- As a good developer, you should count for these unexpected errors, log them and return a proper error to the client.

- Use the Express error middleware to catch any unhandled exceptions in the "request processing pipeline".

- Register the error middleware *after* all the existing routes:

```
app.use(function(err, req, res, next) {
    // Log the exception and return a friendly error to the client.
    res.status(500).send('Something failed');
});
```

- To pass control to the error middleware, wrap your route handler code in a try/catch block and call **next()**.

```
try {
    const genres = await Genre.find();
    ...
}
```

```
catch(ex) {
    next(ex);
});
```

- Adding a try/catch block to every route handler is repetitive and time consuming. Use **express-async-errors** module. This module will monkey-patch your route handlers at runtime. It'll wrap your code within a try/catch block and pass unhandled errors to your error middleware.

- To log errors use **winston.**

- Winston can log errors in multiple transports. A transport is where your log is stored.

- The core transports that come with Winston are **Console**, **File** and **Http**. There are also 3rd-party transports for storing logs in **MongoDB**, **CouchDB**, **Redis** and **Loggly**.

- The error middleware in Express only catches exceptions in the request processing pipeline. Any errors happening during the application startup (eg connecting to MongoDB) will be invisible to Express.

- Use **process.on('uncaughtException')** to catch unhandled exceptions, and **process.on('unhandledRejection')** to catch rejected promises.

- As a best practice, in the event handlers you pass to **process.on(),** you should log the exception and exit the process, because your process may be in an unclean state and it may result in more issues in the future. It's better to restart the process in a clean state. In production, you can use a *process manager* to automatically restart a Node process. You'll learn about that later in the course.