☘️

Part 3

Programming a server with NodeJS and Express

What is included ?

- Node.js and Express
- Deploying app to internet
- Saving data to MongoDB
- Validation and ESlint

a. Node.js and Express

- NodeJS is **a JS runtime** based on Google's **Chrome V8 JS engine**
- Use **interactive node commend prompt** (Node REPL) by typing `node` (Read-eval-print loop)
- browser needs **Babel** because it doesn't support the newest features of JS, this is not the case for Node (running JS in the backend), many latest features are supported **so transpiling is not necessary**
- create template for our application with `npm init` , this will create the file `package.json`
- `package.json` contains information about the project

```
{
    "name": "backend",
    "version": "1.0.0",
    "description": "",
    // entry point of the application
    "main": "index.js",
    // define scripts to run
    "scripts": {
        "start": "node index.js",
    },
    "author": "",
    "license": "ISC"
}
```

- We can define scripts inside package.json and then run them `npm run <script_name>`

Simple web server

```
// code for starting a server
// 'http' is Node's built-in web server module
const http = require('http')

const app = http.createServer((req, res) => {
  // event handler that's called everytime an HTTP request is made to the server
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('Hello World')
})

// binds the http server `app` to listen to HTTP requests sent to port 3001
const PORT = 3001
app.listen(PORT)
console.log(`Server running on port ${PORT}`)
```

- Node uses **CommonJS modules**, it doesn't support **ES6 modules** yet. So we need to change the syntax a little bit (`require('...'), module.exports = {}` instead of (`import ... from '...', export default ...`)
- One key distinction is that **CommonJS** is synchronous, and **ES6 modules** supports async, therefore CommonJS does not work for browsers. (Support for ES6 modules inside Node is coming)
- To send data in **JSON format**, we should change **`Content-Type`** to **`application/json`**, and use **`JSON.stringify(data)`** to send JSON (we need stringify because **JSON is a string, not a JS object**)

Express

- Offers a more pleasant interface to work with than the built-in http module
- install as a project dependency with `npm install express`
- the source code for the dependency (and its dependencies) is installed to the `node_modules` directory
- npm uses the versioning model of **semantic versioning**
- The big idea of **semantic versioning** is that newer versions should be **backwards compatible** if the major number does not change.

Code status	Stage	Rule	Example version
First release	New product	Start with 1.0.0	1.0.0
Backward compatible bug fixes	Patch release	Increment the third digit	1.0.1
Backward compatible new features	Minor release	Increment the middle digit and reset last digit to zero	1.1.0
Changes that break backward compatibility	Major release	Increment the first digit and reset middle and last digits to zero	2.0.0

```
// inside package.json
"dependencies": {
        "express": "^4.17.1"
}
```

The `^` means that the installed version must have the same major version 4 and greater or equal to version 4.17.1

- `npm update` will install the latest dependencies that satisfy the configuration of dependencies in package.json
- `npm install` installs all dependencies defined in package.json (useful if you are working on a project on another computer and `node_modules` is not installed

Web and express

```
// import express
const express = require('express')
// create an express application stored in the app variable
const app = express()

let notes = [
  ...
]

// define 2 routes to the application
// req contains all the info of HTTP request
// res is used to define how the request is responded to
app.get('/', (req, res) => {
  // res.send sends the responding string, express automatically sets the Content-Type header
  // to text/html, the status code is default to 200
  res.send('<h1>Hello World!</h1>')
})

app.get('/api/notes', (req, res) => {
  // res.json sends JSON object, express automatically sets the Content-Type header
  // to application/json
  // JSON.stringify is not necessary, express does it for us
  res.json(notes)
})

const PORT = 3001
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`)
})
```

Nodemon

- When we make changes we need to restart the application to see the changes (cumbersome comparing to React where the browser automatically reloads after changes)
- nodemon watches the files in the directory in which nodemon was started, and it will **automatically restart the node application if any files change**
- `npm install --save-dev nodemon`
- Can be added to devDependencies (**tools that are only needed during the development of the application),** these are not needed when the application is run in production mode on production server.
- We can use nodemon like `node_modules/.bin/nodemon index.js` , or add it as a script and run with `npm run dev` (the path to nodemon is not needed since npm knows to search for nodemon)

```
"scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
  },
```

REST

- **Representational State Transfer**, an architectural style meant for building scalable web applications

- It's a consistent way of defining interfaces that makes it possible for systems to co-operate
- CRUD (Create, read, update, delete)

Fetching a single resource

- Define parameters for routes with the `:`
- parameters are default to be string
- We use `res.status(<status_code>)` to set status

```
// parameters can be accessed through req.params object
app.get("/api/notes/:id", (req, res) => {
  // conver to Number because it is default to be a string
  const id = Number(req.params.id)
  // use find to find the first match in an array, return undefined if not found
  const note = notes.find(note => note.id === id)

  if (note) {
    res.json(note)
  } else {
    // handle note not found
    res.status(404).end()
    // end() is for responding the request without sending any data
```

Deleting resources

- Make **HTTP DELETE** request to the url of the resource
- If deleting is successful, we respond with the **status code 204** (no content) and return no data with the response

```
app.delete('/api/notes/:id', (request, response) => {
  const id = Number(request.params.id)
  notes = notes.filter(note => note.id !== id)

  response.status(204).end()
})
```

Postman

- note that it's not easy to test DELETE from the browser, we can use a tool called **Postman** to make testing the backend easier.
- **The Visual Studio Code REST client**
  - We can use VS Code **REST client plugin** instead of Postman
  - save all REST client request in a directory as files that **end with the .rest extension**

```
// from /requests/get_all_notes.rest
GET http://localhost:3001/api/notes
```

Receiving Data

- We can send data to the server using **POST request**, the data sent is in the **request body** in the JSON format
- In order to access the data easily, we need **express json-parser**

```
app.use(express.json())
```

- json-parser is needed to turn JSON data of a request into JavaScript Object and then attach it to the **body** property of the **request**, else it would be undefined
- access the body using `req.body`
- Server can only parse the data correctly with the **correct request header**. It will not try to guess the format of the data
- Remember to set the correct content type
- We can set the POST request in **REST client** as shown below

```
POST http://localhost:3001/api/notes
Content-Type: application/json

{
    "content": "This is the content",
    "important": false
}
```

- note, we can check the headers in HTTP request by using `req.get()` or by logging `req.headers`

💡 Keep an eye on what's going on in the terminal when working on backend code

- we should check to make sure that some attributes is passed in inside req.body

```
const generateId = () => {
  const maxId = notes.length > 0
    ? Math.max(...notes.map(n => n.id))
    : 0
  return maxId + 1
}

app.post('/api/notes', (request, response) => {
  const body = request.body

  if (!body.content) {
    // we need to `return` here so the code stops executing afterwards
    return response.status(400).json({
      error: 'content missing'
    })
  }

  const note = {
    content: body.content,
    important: body.important || false,
    date: new Date(),
    id: generateId(),
  }

  notes = notes.concat(note)

  response.json(note)
})
```

- `notes.map(n => n.id)` creates an array of ids. However, `Math.max` takes in numbers separated by commas, so we need to **spread** the array using the `...` operator

About HTTP request types

- We should handle HTTP requests in a way such that they have the property of **safety** and **idempotence**
- Safety: executing request must not cause any side-effects in the server
- Idempotence: the result should be the same regardless of how many times a request is sent
- POST is the only HTTP request type that is neither safe nor idempotent

Middleware

- **Middleware:** functions that can be used for handling **request** and **response** objects
- json-parser is an example of a **middleware** (it takes the raw data from request, parse it into a JS object and assign it to the request object as a new property `body`)
- We may use several middleware at the same time. When we have more than one, they're executed one by one in the order they were taken into use in express

```
const requestLogger = (req, res, next) => {
  console.log('Method:', req.method)
  console.log('Path:  ', req.path)
  console.log('Body:  ', req.body)
  console.log('---')
  next()
}
```

- As shown above, middleware receives 3 parameters: **request, response, next**
- `next()` should be called at the end of the function, it yields control to the next middleware
- The middleware can be used like `app.use(requestLogger)`
- We usually define middlewares before our routes (so middlewares are executed before route event handler)
- Defining middlewares after our routes means they will be called when no route handles the HTTP request

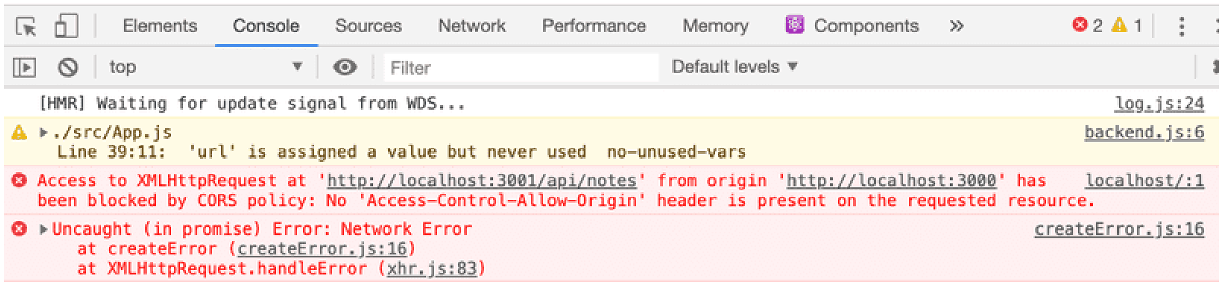
```
// after all routes
const unknownEndpoint = (req, res) => {
  res.status(404).send({ error: 'unknown endpoint' })
}
app.use(unknownEndpoint);
```

- The middleware `morgan` is good for logging HTTP request information

b. Deploying app to internet

Same origin policy and CORS (Cross-Origin Resource Sharing)

- We cannot connect the frontend and the backend simply by making request to our backward server, it gives the following error



```
[HMR] Waiting for update signal from WDS...                                          log.js:24
./src/App.js                                                                      backend.js:6
  Line 39:11:  'url' is assigned a value but never used  no-unused-vars
Access to XMLHttpRequest at 'http://localhost:3001/api/notes' from origin 'http://localhost:3000' has    localhost/:1
been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.
Uncaught (in promise) Error: Network Error                                        createError.js:16
    at createError (createError.js:16)
    at XMLHttpRequest.handleError (xhr.js:83)
```

💡 **Same origin**: same port, host, and protocol

- By default, JS code of an application that runs in a browser can only communicate with a server in the same origin. We have a server running in a different port (different origin), therefore access is forbidden
- We can allow requests from another origin by using the CORS mechanics (allow restricted resources to be requested even if origin is different)
- We can enable CORS by using a middleware called `cors` (install it first)

```
const cors = require('cors');
app.use(cors());
```

Application to the Internet

- We will use **Heroku** for first deploying the backend to the internet

💡 Make sure to install Heroku CLI (command line interface) first

0. Login to heroku with `heroku login`

1. Add `Procfile` file at root to tell Heroku how to start the application
    - `web: npm start`
2. Change the definition of the port for our application

```
const PORT = process.env.PORT || 3001
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`)
});
```

    - Heroku configures port based on environment variables, so we try to use that if it's defined
3. The application must be a git repository, remember to ignore `node_modules` in `.gitignore`
4. Create Heroku app with `heroku create` , then move committed code to Heroku with `git push heroku master`

💡 Show Heroku logs via `heroku logs -t` (this includes any console.log())

Frontend production build

- We've been running React in **development mode**, this is configured for clear error messages, immediately render code changes to browser, etc.
- When the app is deployed, we must create a **production build**, which is configured for optimized production
- create production build by running `npm run build` with React apps created with `create-react-app`
    - This creates a directory `build` which contains the directory `static`
    - HTML files are in build, minified JS code is in static

    💡 all JS will be minified into one file

Serving static files from the backend

- One option is to copy the production build `build` directory to the root of the backend repository and configure the backend to show frontend's main page `build/index.html` as its main page
- express needs another middleware `static` to serve static content (our production build) `app.use(express.static('build'))`
- this will put both the frontend and the backend at the same address, so we can fetch a **relative URL** from the frontend, (e.g. `/api/persons`)

Streamlining deploying of the frontend

```
{
  "scripts": {
    //...
    "build:ui": "rm -rf build && cd <directory for frontend> && npm run build && cp -r build <directory for backend>",
    "deploy": "git push heroku master",
    "deploy:full": "npm run build:ui && git add . && git commit -m uibuild && npm run deploy",
    "logs:prod": "heroku logs --tail"
  }
}
```

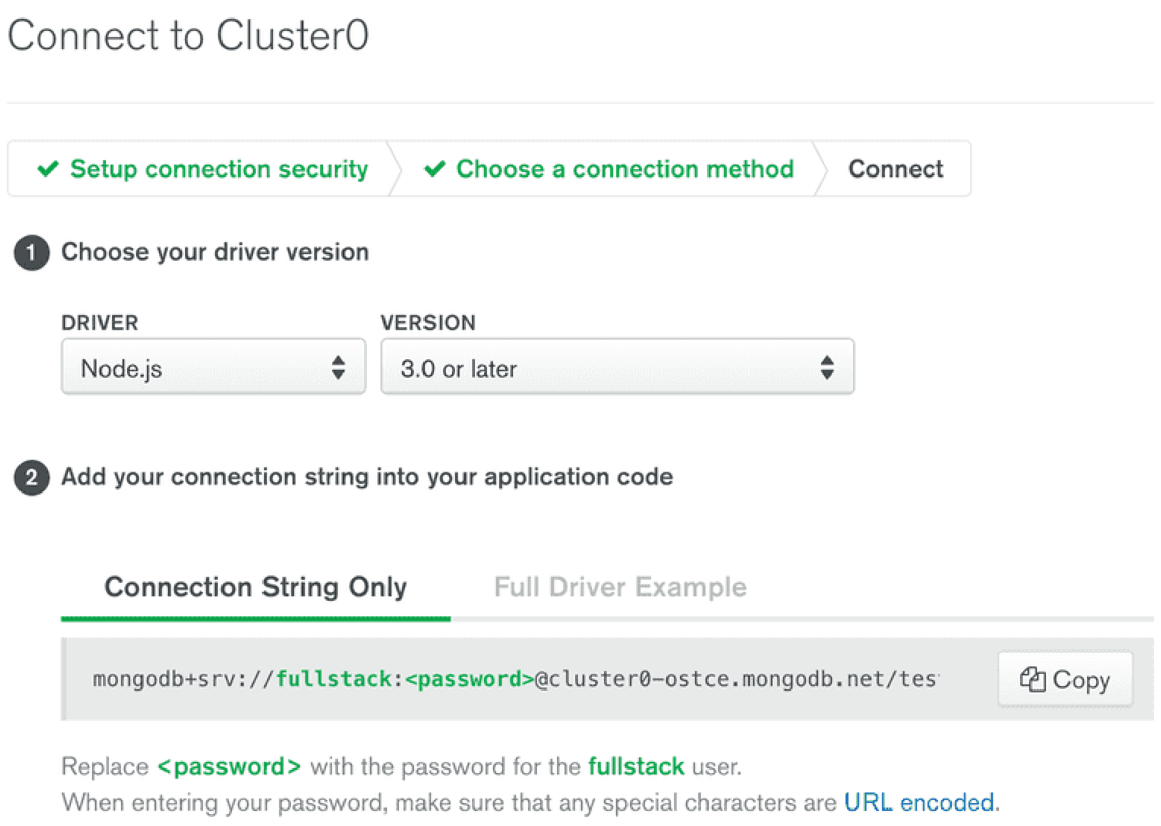
- build:ui builds the frontend and replace the build file in backend
- deploy deploys the committed code to heroku
- deploy:full combines all process (build ui, commit code, deploy)
- logs:prod shows the heroku logs

Proxy

- We changed the frontend to use relative path to fetch data from the backend, this will cause problem when we are running only the frontend
- For projects created with `create-react-app` , we can solve this by adding a proxy to package.json
- `"proxy": "http://localhost:3001"`
- This will make the react development environment work as a proxy, if the React code does an HTTP request to localhost:3000 not managed by the React application itself, the request will be redirected to localhost:3001

c. Saving data to MongoDB

- Without a database, our changes to the server is reseted when we restart or crash the server, we need a database to store data indefinitely.
- **MongoDB** is a **document database (NoSQL),** differ from **relational databases (SQL)**
- We will use **MongoDB Atlas** which provides Mongo database cloud service


Connect to Cluster0

✓ Setup connection security ✓ Choose a connection method Connect

1 Choose your driver version

DRIVER: Node.js

VERSION: 3.0 or later

2 Add your connection string into your application code

Connection String Only | Full Driver Example

```
mongodb+srv://fullstack:<password>@cluster0-ostce.mongodb.net/tes
```

Copy

Replace <password> with the password for the fullstack user.
When entering your password, make sure that any special characters are URL encoded.

- Follow the instruction on MongoDB Atlas (Create cluster → Create collection → Connect), it will show you the **MongoDB URI** to use (the address of the database)

💡 Note: remember to create a user to use the database (etc. username: **fullstack,** password: **fullstack**)

- We could use the DB with the **official MongoDb Node.js driver** library, however, we will use **Mongoose** library which offers a higher level API

💡 This is similar to how we use **express** over **http** library

- Mongoose is an **object document mapper (ODM)**, it allows us to save JS objects as Mongo documents easily
- `npm install mongoose`

```
const mongoose = require('mongoose');

// we can access command line arguments with `process.argv`
if (process.argv.length < 3) {
  console.log('Please provide the password as an argument: node mongo.js <password>');
  // exit the current node process (stop program execution)
  process.exit(1);
}

const password = process.argv[2];

const url = < Insert MongoURI here >

// this line establish the database connection
mongoose.connect(url, { useNewUrlParser: true, useUnifiedTopology: true, useFindAndModify: false, useCreateIndex: true });

// define the schema of a note
const noteSchema = new mongoose.Schema({
  content: String,
  date: Date,
  important: Boolean,
});

// define model
const Note = mongoose.model('Note', noteSchema);

// model is used to create a object of that model
const note = new Note({
  content: 'HTML is Easy',
  date: new Date(),
  important: true,
});

note.save().then(result => {
  console.log('note saved!');
  mongoose.connection.close();
}
```

- A **schema** tells Mongoose how objects are to be stored in the database
- The model takes two parameter, the first is a **singular name of the model,** the matching name of the collection will be the lowercased plural `notes` . The second is the schema for the model to follow

💡 Mongo itself does not care about the structure of data (**schemaless)**, Mongoose allows us to define schemas at the application level to give shape of the documents stored in the database

- think of models as **constructor functions** that create new JS objects that contains properties of that model
- Save objects to that database with the `save` method, and event handler can be provided with the `then` method (it'll be called when object is saved)

💡 close database connection inside event handler or else the program will never finish execution

- The `mongoose.connection.close()` must be put inside callback function or else it will get called before the database operation is done.

BAD EXAMPLE

```
Person
  .find({})
  .then(persons=> {
    // ...
    // the correct place to put mongoose.connection.close() is here
  })

// this is ran immediately, and Person.find will never finish (connection is closed)
mongoose.connection.close()
```

Fetching objects from the database

```
Note.find({}).then(result => {
  result.forEach(note => {
    console.log(note)
  })
  mongoose.connection.close()
})
```

- we use the `find` method of the `Note` model. The parammeter of the method is an object expressing search condition (empty `{}` finds all objects inside collection)

💡 DO NOT include the password in files that you commit and push to GitHub

Database configuration into its own module

- We can separate DB configuration into its own module and only export the necessary info
- We can format the objects returned by mongoose by modifying the `toJSON` method of the schema

💡 Note: the objects inside database is still the same, the transformation is applied when objects is turned into the JSON format (e.g. `response.json(<object>`)

- exporting with **commonJS** is done with `module.exports = <the value to export>`

```
const mongoose = require('mongoose')

// using dotenv
const url = process.env.MONGODB_URI
console.log('connecting to', url)

// the connect method also handles callback through the `then` method
mongoose.connect(url, { useNewUrlParser: true, useUnifiedTopology: true, useFindAndModify: false, useCreateIndex: true })
  .then(result => {    console.log('connected to MongoDB')  })
  .catch((error) => {    console.log('error connecting to MongoDB:', error.message)  })

const noteSchema = new mongoose.Schema({
  content: String,
  date: Date,
  important: Boolean,
})

// modify the objects returned by Mongoose
noteSchema.set('toJSON', {
  transform: (document, returnedObject) => {
    returnedObject.id = returnedObject._id.toString()
    delete returnedObject._id
    delete returnedObject.__v
  }
})

module.exports = mongoose.model('Note', noteSchema)
```

dotenv

- It's a good idea to make MongoURI a environment variable, we can define environment variables via `dotenv` library
- `npm install dotenv` , then we create a `.env` file at the root of the project, then define envrionment varaibles inside the file like `PORT=3001`

💡 Make sure to gitignore `.env` right away, we do not want to publish confidential information to git

- To use the environment variables inside `.env` , we put `require('dotenv').config()` at the **top of index.js**. Then we can reference them like normal environment variables like `process.env.<variable name>`

💡 Note that we are not closing the mongoDB connection, this is because our server is persistent and needs to maintain the connection. The connection will close when we stop the server

- The environment variables here will only be used when the backend is not in production mode (i.e. Heroku)
  - We need to set environment variables in heroku with `heroku config:set`

Error handling

- add `catch` blocks to handle errors (when the returned promise is **rejected**)

💡 Example error: id provided doesn't match the format of _id of a MongoDB object (this will throw **CastError**)

```
app.get('/api/notes/:id', (request, response) => {
  Note.findById(request.params.id)
    .then(note => {
      if (note) {
        response.json(note)
      } else {
        response.status(404).end()
      }
    })
    .catch(error => {
      console.log(error)
      response.status(400).send({ error: 'malformatted id' })    })
})
```

- in the above code, if the format of id is invalid, code will throw a CastError, which will be catched in the `catch` method to handle.

- It's a good idea to log `error` directly to debug

Moving error handling into middleware

- It's a good idea because it groups all error handlings into one single place
- to use a error handling middleware (**special kind of middleware**), we pass the error forward with the `next` function

```
// to use next, pass it in as the third parameter to the handler
app.get('/api/notes/:id', (request, response, next) => {
  Note.findById(request.params.id)
    .then(note => {
      if (note) {
        response.json(note)
      } else {
        response.status(404).end()
      }
    })
    .catch(error => next(error))
})
```

- When `next` is called with a parameter, the execution will continue to the **error handler middlware**, else it would just move onto the next route or middleware
- Express **error handler middleware** is defined with a function that accepts four parameters

```
const errorHandler = (error, request, response, next) => {
  console.log(error.message)
  if (error.name === 'CastError') {
    // since we are sending a JSON directly, we don't need to use `.json()`, `.send()` is OK
    return response.status(400).send({error: 'malformatted id' })
  }

  next(error)
}
app.use(errorHandler)
```

- the `next(error)` inside our error handler will pass the error forward to the **default Express error handler**

The order of middleware loading

- It's best to place error handler middleware last inside the file

```
app.use(express.static('build'))
app.use(express.json())
app.use(logger)

app.post('/api/notes', (request, response) => {
  const body = request.body
  // ...
})

const unknownEndpoint = (request, response) => {
  response.status(404).send({ error: 'unknown endpoint' })
}
```

```
// handler of requests with unknown endpoint
app.use(unknownEndpoint)

const errorHandler = (error, request, response, next) => {
  // ...
}

// handler of requests with result to errors
app.use(errorHandler)
```

Other operations

- delete with `findByIdAndRemove`
    - respond with status code **204 no content** when the deletion is successful or the id does not exist

```
app.delete('/api/notes/:id', (request, response, next) => {
  Note.findByIdAndRemove(request.params.id)
    .then(result => {
      response.status(204).end()
    })
    .catch(error => next(error))
})
```

- update objects with `findByIdAndUpdate`

```
app.put('/api/notes/:id', (request, response, next) => {
  const body = request.body

  const note = {
    content: body.content,
    important: body.important,
  }

  Note.findByIdAndUpdate(request.params.id, note, { new: true })
    .then(updatedNote => {
      response.json(updatedNote)
    })
    .catch(error => next(error))
})
```

- add the `{new: true}` parameter so updatedNote will be the new modified document instead of the original

d. Validation and ESLint

- One way to validate the format of the data is to use the **validation** functionality in Mongoose
- We can define specific validation rules for each field in the schema

```
const noteSchema = new mongoose.Schema({
  content: {
    type: String,
    minlength: 5,
    required: true
  },
  date: {
    type: Date,
    required: true
  },
  important: Boolean
})
```

- validators like `minlength` and `required` are built-in of Mongoose, we can also use **customer validators** if necessary
- when we try to store an object that breaks one of the contraint, the operation will throw an exception, we can catch it with our error handler middleware as follow

```
const errorHandler = (error, request, response, next) => {
  console.error(error.message)

  if (error.name === 'CastError') {
    return response.status(400).send({ error: 'malformatted id' })
  } else if (error.name === 'ValidationError') {
    return response.status(400).json({ error: error.message })
  }

  next(error)
}
```

Promise Chaining

- the `then` method of a promise returns a promise and we can access the returned data with another `then` method, this along with the simplicity of compact arrow function syntax can help us clean up code

```
app.post('/api/notes', (request, response, next) => {
  // ...

  note
    .save()
    .then(savedNote => savedNote.toJSON())
    .then(savedAndFormattedNote => {
      response.json(savedAndFormattedNote)
    })
    .catch(error => next(error))
})
```

- The example given above is contrived, in reality we can just use `response.json` because it automatically call the `toJSON` function of the data passed in

From exercises

- enable validation for unique values by installing the `mongoose-unique-validator` package
- the frontend will crash when an error (e.g. status 404) happen, we can display error message instead of crashing by catching the errors

```
service
  .create({ name: newName, phone: newNumber })
  .then((res) => {
      setPersons(persons.concat(res));
```

```
        setMessage({ message: `added ${newName}`, type: 'success' });
    })
    .catch((error) => {
        setMessage({
            message: error.response.data.error,
            type: 'error',
        });
    });
```

- we can access the error message with `error.response.data` , we used an extra `.error` because we know our server responded with a JSON of structure `{ error: <error message> }`
- On update operations, mongoose validators are off by default, turn them on by setting the `runValidators` option as follow

```
const opts = { runValidators: true };
Toy.updateOne({}, { color: 'not a color' }, opts, function(err) {
  assert.equal(err.errors.color.message,
    'Invalid color');
});
```

Lint

- Lint or a linter is a tool that detects and flags errors in programming languages, including stylistic errors.
- The leading tool for static analysis (linting) for JS is **ESlint** `npm install eslint --save-dev`
- we can initialize a default ESlint configuration with the command `node_modules/.bin/eslint --init`
- configurations are saved in `.eslintrc.js` file
- inspecting and validating a file can be done like `node_modules/.bin/eslint index.js` (we can create a npm script for this)
- we don't want eslint to check `build` directory, we can ignore it by creating a `.eslintignore` file (works like `.gitignore`

- We can also use the VSCode extension for Eslint which runs the linter continuously and underline violations with red line
- there are existing eslint configurations so it might be a good idea to adopt a ready-made configurations from someone else