

Validation and ESLint

There are usually constraints that we want to apply to the data that is stored in our application's database. Our application shouldn't accept notes that have a missing or *empty content* property. The validity of the note is checked in the route handler:

```
app.post('/api/notes', (request, response) => {
  const body = request.body
  if (body.content === undefined) {
    return response.status(400).json({ error: 'content missing' })
  }

  // ...
})
```

If the note does not have the *content* property, we respond to the request with the status *code 400 bad request*.

One smarter way of validating the format of the data before it is stored in the database is to use the [validation](#) functionality available 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
  },
  important: Boolean
})
```

The *content* field is now required to be at least five characters long and it is set as required, meaning that it can not be missing. We have not added any constraints to the *important* field, so its definition in the schema has not changed.

The *minLength* and *required* validators are [built-in](#) and provided by Mongoose. The Mongoose [custom validator](#) functionality allows us to create new validators if none of the built-in ones cover our needs.

If we try to store an object in the database that breaks one of the constraints, the operation will throw an exception. Let's change our handler for creating a new note so that it passes any potential exceptions to the error handler middleware:

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

```

const note = new Note({
  content: body.content,
  important: body.important || false,
})

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

```

Let's expand the error handler to deal with these validation errors:

```

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)
}

```

When validating an object fails, we return the following default error message from Mongoose:

We notice that the backend has now a problem: validations are not done when editing a note. The [documentation](#) addresses the issue by explaining that validations are not run by default when *findOneAndUpdate* and related methods are executed.

The fix is easy. Let us also reformulate the route code a bit:

```

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

  Note.findByIdAndUpdate(
    request.params.id,
    { content, important },
    { new: true, runValidators: true, context: 'query' }
  )
    .then(updatedNote => {
      response.json(updatedNote)
    })
    .catch(error => next(error))
})

```

Deploying the database backend to production

The application should work almost as-is in Fly.io/Render. We do not have to generate a new production build of the frontend since changes thus far were only on our backend.

The environment variables defined in dotenv will only be used when the backend is not in production mode, i.e. Fly.io or Render.

For production, we have to set the database URL in the service that is hosting our app.

In Fly.io that is done `fly secrets set`:

```
fly secrets set
MONGODB_URI='mongodb+srv://fullstack:thepasswordishere@cluster0.o1opl.mongodb.net/noteApp?retryWrites=true&w=majority'
```

When the app is being developed, it is more than likely that something fails. Eg. when I deployed my app for the first time with the database, not a single note was seen:

The network tab of the browser console revealed that fetching the notes did not succeed, the request just remained for a long time in the `pending` state until it failed with status code 502.

The browser console has to be open *all the time*!

It is also vital to follow continuously the server logs. The problem became obvious when the logs were opened with `fly logs`:

The database url was `undefined`, so the command `fly secrets set MONGODB_URI` was forgotten.

You will also need to whitelist the fly.io app's IP address in MongoDB Atlas. If you don't MongoDB will refuse the connection.

Sadly, fly.io does not provide you a dedicated IPv4 address for your app, so you will need to allow all IP addresses in MongoDB Atlas.

When using Render, the database url is given by defining the proper env in the dashboard:

The Render Dashboard shows the server logs:

You can find the code for our current application in its entirety in the part3-6 branch of [this GitHub repository](#).

Exercises 3.19. - 3.21.

3.19*: Phonebook database, step 7

Expand the validation so that the name stored in the database has to be at least three characters long.

Expand the frontend so that it displays some form of error message when a validation error occurs. Error handling can be implemented by adding a `catch` block as shown below:

```
personService
  .create({ ... })
  .then(createdPerson => {
    // ...
  })
  .catch(error => {
    // this is the way to access the error message
    console.log(error.response.data.error)
  })
```

You can display the default error message returned by Mongoose, even though they are not as readable as they could be: **NB:** On update operations, mongoose validators are off by default. [Read the documentation](#) to determine how to enable them.

3.20*: Phonebook database, step 8

Add validation to your phonebook application, which will make sure that phone numbers are of the correct form. A phone number must:

- have length of 8 or more
- be formed of two parts that are separated by -, the first part has two or three numbers and the second part also consists of numbers
- - eg. 09-1234556 and 040-22334455 are valid phone numbers
- - eg. 1234556, 1-22334455 and 10-22-334455 are invalid Use a [Custom validator](#) to implement the second part of the validation.

If an HTTP POST request tries to add a person with an invalid phone number, the server should respond with an appropriate status code and error message.

3.21 Deploying the database backend to production

Generate a new "full stack" version of the application by creating a new production build of the frontend, and copying it to the backend repository. Verify that everything works locally by using the entire application from the address <http://localhost:3001/>.

Push the latest version to Fly.io/Render and verify that everything works there as well.

NOTE: you should deploy the BACKEND to the cloud service. If you are using Fly.io the commands should be run in the root directory of the backend (that is, in the same

directory where the backend package.json is). In case of using Render, the backend must be in the root of your repository.

You shall NOT be deploying the frontend directly at any stage of this part. It is just backend repository that is deployed throughout the whole part, nothing else.

Lint

Before we move on to the next part, we will take a look at an important tool called [lint](#). Wikipedia says the following about lint:

Generically, lint or a linter is any tool that detects and flags errors in programming languages, including stylistic errors. The term lint-like behavior is sometimes applied to the process of flagging suspicious language usage. Lint-like tools generally perform static analysis of source code.

In compiled statically typed languages like Java, IDEs like NetBeans can point out errors in the code, even ones that are more than just compile errors. Additional tools for performing [static analysis](#) like [checkstyle](#), can be used for expanding the capabilities of the IDE to also point out problems related to style, like indentation.

In the JavaScript universe, the current leading tool for static analysis (aka "linting") is [ESLint](#).

Let's install ESLint as a development dependency to the notes backend project with the command:

```
npm install eslint @eslint/js --save-dev
```

After this we can initialize a default ESLint configuration with the command:

```
npx eslint --init
```

We will answer all of the questions:

The configuration will be saved in the generated `eslint.config.mjs` file:

```
import globals from 'globals'

export default [
  { files: ["**/*.js"], languageOptions: {sourceType: "commonjs"} },
  { languageOptions: { globals: globals.browser } },
]
```

We will reformat the configuration file from its current form to the following:

```
// ...
export default [
  {
```

```

files: ["**/*.js"],
languageOptions: {
  sourceType: "commonjs",
  globals: {
    ...globals.node,
  },
  ecmaVersion: "latest",
},
},
]

```

So far, our ESLint configuration file defines the `files` option with `["**/*.js"]`, which tells ESLint to look at all JavaScript files in our project folder. The `languageOptions` property specifies options related to language features that ESLint should expect, in which we defined the `sourceType` option as `"commonjs"`. This indicates that the JavaScript code in our project uses the CommonJS module system, allowing ESLint to parse the code accordingly.

The `globals` property specifies global variables that are predefined. The spread operator applied here tells ESLint to include all global variables defined in the `globals.node` settings such as the `process`. In the case of browser code we would define here `globals.browser` to allow browser specific global variables like `window`, and `document`.

Finally, the `ecmaVersion` property is set to `"latest"`. This sets the ECMAScript version to the latest available version, meaning ESLint will understand and properly lint the latest JavaScript syntax and features.

We want to make use of [ESLint's recommended](#) settings along with our own. The `@eslint/js` package we installed earlier provides us with predefined configurations for ESLint. We'll import it and enable it in the configuration file:

```

// ...
import js from '@eslint/js'
// ...

export default [
  js.configs.recommended, // highlight-line
  {
    // ...
  }
]

```

We've added the `js.configs.recommended` to the top of the configuration array, this ensures that ESLint's recommended settings are applied first before our own custom options.

Let's continue building the configuration file. Install a [plugin](#) that defines a set of code style-related rules:

```
npm install --save-dev @stylistic/eslint-plugin-js
```

Import and enable the plugin, and add these four code style rules:

```
// ...
import stylisticJs from '@stylistic/eslint-plugin-js'

export default [
  {
    // ...
    plugins: {
      '@stylistic/js': stylisticJs
    },
    rules: {
      '@stylistic/js/indent': [
        'error',
        2
      ],
      '@stylistic/js/linebreak-style': [
        'error',
        'unix'
      ],
      '@stylistic/js/quotes': [
        'error',
        'single'
      ],
      '@stylistic/js/semi': [
        'error',
        'never'
      ],
    },
  },
]
```

The `plugins` property provides a way to extend ESLint's functionality by adding custom rules, configurations, and other capabilities that are not available in the core ESLint library. We've installed and enabled the `@stylistic/eslint-plugin-js`, which adds JavaScript stylistic rules for ESLint. In addition, rules for indentation, line breaks, quotes, and semicolons have been added. These four rules are all defined in the [Eslint styles plugin](#).

Inspecting and validating a file like `index.js` can be done with the following command:

```
npx eslint index.js
```

It is recommended to create a separate `npm script` for linting:

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

```
// ...
"lint": "eslint ."
},
// ...
}
```

Now the `npm run lint` command will check every file in the project.

Files in the `dist` directory also get checked when the command is run. We do not want this to happen, and we can accomplish this by adding an object with the `ignores` property that specifies an array of directories and files we want to ignore.

```
// ...
export default [
  // ...
  {
    ignores: ["dist/**"],
  },
  //...
]
```

This causes the entire `dist` directory to not be checked by ESLint.

Lint has quite a lot to say about our code:

Let's not fix these issues just yet.

A better alternative to executing the linter from the command line is to configure an `eslint-plugin` to the editor, that runs the linter continuously. By using the plugin you will see errors in your code immediately. You can find more information about the Visual Studio ESLint plugin [here](#).

The VS Code ESLint plugin will underline style violations with a red line:

This makes errors easy to spot and fix right away.

ESLint has a vast array of [rules](#) that are easy to take into use by editing the `eslint.config.mjs` file.

Let's add the `eqeqeq` rule that warns us if equality is checked with anything but the triple equals operator. The rule is added under the `rules` field in the configuration file.

```
export default [
  // ...
  rules: {
    // ...
    'eqeqeq': 'error',
  },
]
```



```
    },  
  ]  
}
```

While we're at it, let's make a few other changes to the rules.

Let's prevent unnecessary [trailing spaces](#) at the ends of lines, require that [there is always a space before and after curly braces](#), and also demand a consistent use of whitespaces in the function parameters of arrow functions.

```
export default [  
  // ...  
  rules: {  
    // ...  
    'eqeqeq': 'error',  
    'no-trailing-spaces': 'error',  
    'object-curly-spacing': [  
      'error', 'always'  
    ],  
    'arrow-spacing': [  
      'error', { 'before': true, 'after': true },  
    ],  
  },  
]
```

Our default configuration takes a bunch of predefined rules into use from:

```
// ...  
export default [  
  js.configs.recommended,  
  // ...  
]
```

This includes a rule that warns about `console.log` commands. Disabling a rule can be accomplished by defining its "value" as 0 or "off" in the configuration file. Let's do this for the `no-console` rule in the meantime.

```
[  
  {  
    // ...  
    rules: {  
      // ...  
      'eqeqeq': 'error',  
      'no-trailing-spaces': 'error',  
      'object-curly-spacing': [  
        'error', 'always'  
      ],  
      'arrow-spacing': [  

```

```

    'error', { 'before': true, 'after': true },
  ],
  'no-console': 'off',
},
},
]

```

Disabling the no-console rule will allow us to use console.log statements without ESLint flagging them as issues. This can be particularly useful during development when you need to debug your code. Here's the complete configuration file with all the changes we have made so far:

```

import globals from "globals";
import stylisticJs from '@stylistic/eslint-plugin-js'
import js from '@eslint/js'

export default [
  js.configs.recommended,
  {
    files: ["**/*.js"],
    languageOptions: {
      sourceType: "commonjs",
      globals: {
        ...globals.node,
      },
      ecmaVersion: "latest",
    },
    plugins: {
      '@stylistic/js': stylisticJs
    },
    rules: {
      '@stylistic/js/indent': [
        'error',
        2
      ],
      '@stylistic/js/linebreak-style': [
        'error',
        'unix'
      ],
      '@stylistic/js/quotes': [
        'error',
        'single'
      ],
      '@stylistic/js/semi': [
        'error',
        'never'
      ],
      'eqeqeq': 'error',
      'no-trailing-spaces': 'error',
    }
  }
]

```

```

    'object-curly-spacing': [
      'error', 'always'
    ],
    'arrow-spacing': [
      'error', { 'before': true, 'after': true },
    ],
    'no-console': 'off',
  },
},
{
  ignores: ["dist/**", "build/**"],
},
]

```

NB when you make changes to the `eslint.config.mjs` file, it is recommended to run the linter from the command line. This will verify that the configuration file is correctly formatted:

terminal output from `npm run lint`

If there is something wrong in your configuration file, the lint plugin can behave quite erratically.

Many companies define coding standards that are enforced throughout the organization through the ESLint configuration file. It is not recommended to keep reinventing the wheel over and over again, and it can be a good idea to adopt a ready-made configuration from someone else's project into yours. Recently many projects have adopted the Airbnb [Javascript style guide](#) by taking Airbnb's [ESLint](#) configuration into use.

You can find the code for our current application in its entirety in the part3-7 branch of [this GitHub repository](#).

Exercise 3.22.

3.22: Lint configuration

Add ESLint to your application and fix all the warnings.

This was the last exercise of this part of the course. It's time to push your code to GitHub and mark all of your finished exercises to the [exercise submission system](#).