



## d 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
  },
  date: {
    type: Date,
    required: true
  },
})
```

```
    important: Boolean
  })
```

The *content* field is now required to be at least five characters long. The *date* field is set as required, meaning that it can not be missing. The same constraint is also applied to the *content* field, since the minimum length constraint allows the field to 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,
    date: new Date(),
  })

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

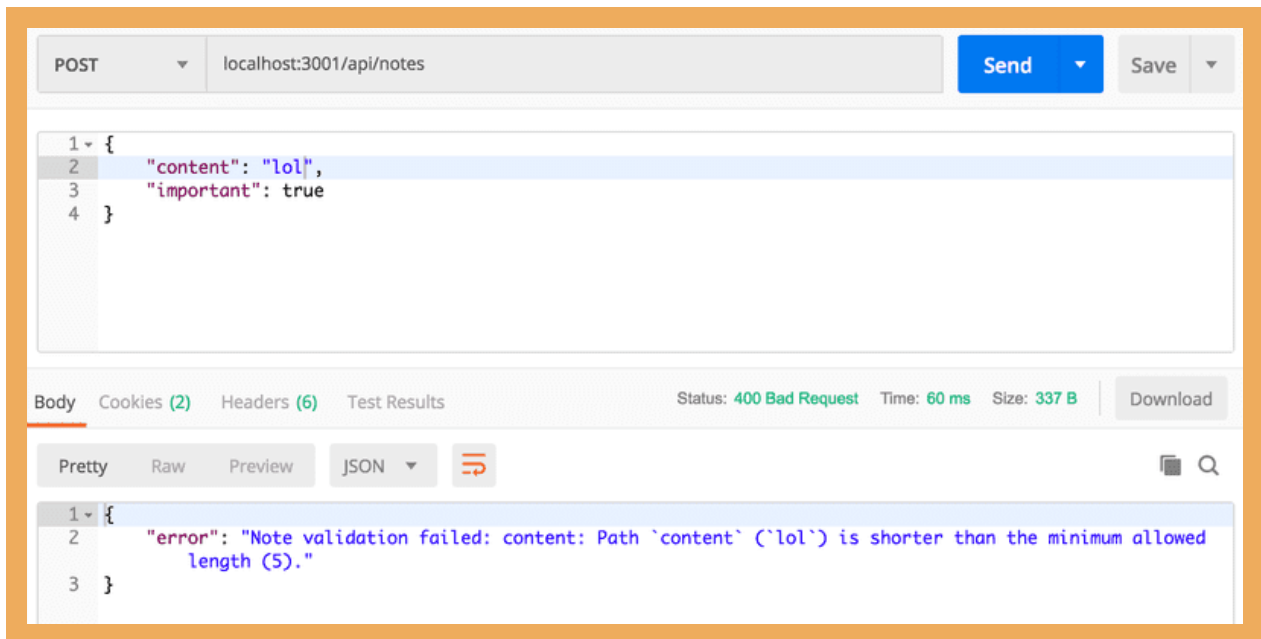
Let's expand the error handler to deal with these validation errors:

```
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:



## Promise chaining

Many of the route handlers changed the response data into the right format by implicitly calling the `toJSON` method from `response.json`. For the sake of an example, we can also perform this operation explicitly by calling the `toJSON` method on the object passed as a parameter to `then`:

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

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

We can accomplish the same functionality in a much cleaner way with promise chaining:

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

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

```
  })
  .then(savedAndFormattedNote => {
    response.json(savedAndFormattedNote)
  })
  .catch(error => next(error))
})
```

In the first `then` we receive `savedNote` object returned by Mongoose and format it. The result of the operation is returned. Then as we discussed earlier, the `then` method of a promise also returns a promise and we can access the formatted note by registering a new callback function with the `then` method.

We can clean up our code even more by using the more compact syntax for arrow functions:

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

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

In this example, Promise chaining does not provide much of a benefit. The situation would change if there were many asynchronous operations that had to be done in sequence. We will not dive further into the topic. In the next part of the course we will learn about the *async/await* syntax in JavaScript, that will make writing subsequent asynchronous operations a lot easier.

## Deploying the database backend to production

The application should work almost as-is in Heroku. We do have to generate a new production build of the frontend due to the changes that we have made to our frontend.

The environment variables defined in `dotenv` will only be used when the backend is not in *production mode*, i.e. Heroku.

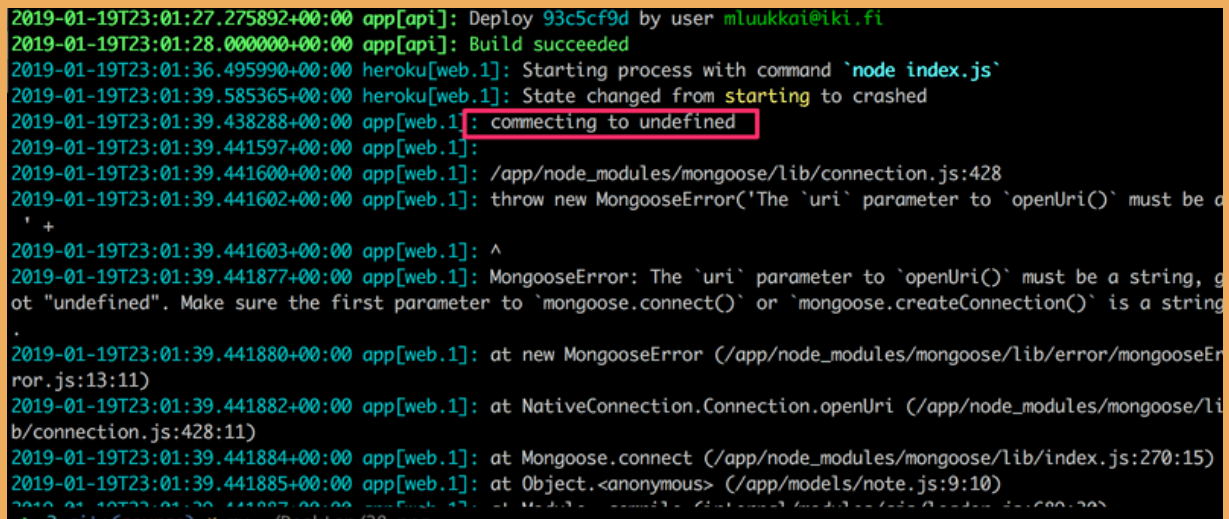
We defined the environment variables for development in file `.env`, but the environment variable that defines the database URL in production should be set to Heroku with the `heroku config:set` command.

```
$ heroku config:set MONGODB_URI=mongodb+srv://fullstack:secretpasswordhere@cluster0-ostce.mongodb.net/
```

NB: if the command causes an error, give the value of MONGODB\_URI in apostrophes:

```
$ heroku config:set MONGODB_URI='mongodb+srv://fullstack:secretpasswordhere@cluster0-ostce.mongodb.net,'
```

The application should now work. Sometimes things don't go according to plan. If there are problems, *heroku logs* will be there to help. My own application did not work after making the changes. The logs showed the following:



```
2019-01-19T23:01:27.275892+00:00 app[api]: Deploy 93c5cf9d by user mluukkai@iki.fi
2019-01-19T23:01:28.000000+00:00 app[api]: Build succeeded
2019-01-19T23:01:36.495990+00:00 heroku[web.1]: Starting process with command `node index.js`
2019-01-19T23:01:39.585365+00:00 heroku[web.1]: State changed from starting to crashed
2019-01-19T23:01:39.438288+00:00 app[web.1]: connecting to undefined
2019-01-19T23:01:39.441597+00:00 app[web.1]:
2019-01-19T23:01:39.441600+00:00 app[web.1]: /app/node_modules/mongoose/lib/connection.js:428
2019-01-19T23:01:39.441602+00:00 app[web.1]: throw new MongooseError('The `uri` parameter to `openUri()` must be a
2019-01-19T23:01:39.441603+00:00 app[web.1]: ^
2019-01-19T23:01:39.441877+00:00 app[web.1]: MongooseError: The `uri` parameter to `openUri()` must be a string, g
2019-01-19T23:01:39.441880+00:00 app[web.1]: at new MongooseError (/app/node_modules/mongoose/lib/error/mongooseEr
2019-01-19T23:01:39.441882+00:00 app[web.1]: at NativeConnection.Connection.openUri (/app/node_modules/mongoose/li
2019-01-19T23:01:39.441884+00:00 app[web.1]: at Mongoose.connect (/app/node_modules/mongoose/lib/index.js:270:15)
2019-01-19T23:01:39.441885+00:00 app[web.1]: at Object.<anonymous> (/app/models/note.js:9:10)
2019-01-19T23:01:39.441887+00:00 app[web.1]: at Model.<anonymous> (/app/models/note.js:9:10)
2019-01-19T23:01:39.441888+00:00 app[web.1]: at Model.<anonymous> (/app/models/note.js:9:10)
```

For some reason the URL of the database was undefined. The *heroku config* command revealed that I had accidentally defined the URL to the `MONGO_URL` environment variable, when the code expected it to be in `MONGODB_URI`.

You can find the code for our current application in its entirety in the *part3-5* branch of [this github repository](#).

## Exercises 3.19.-3.21.

### 3.19: Phonebook database, step7

Add validation to your phonebook application, that will make sure that a newly added person has a unique name. Our current frontend won't allow users to try and create duplicates, but we can attempt to create them directly with Postman or the VS Code REST client.

Mongoose does not offer a built-in validator for this purpose. Install the [mongoose-unique-validator](#) package with npm and use it instead.

If an HTTP POST request tries to add a name that is already in the phonebook, the server must respond with an appropriate status code and error message.

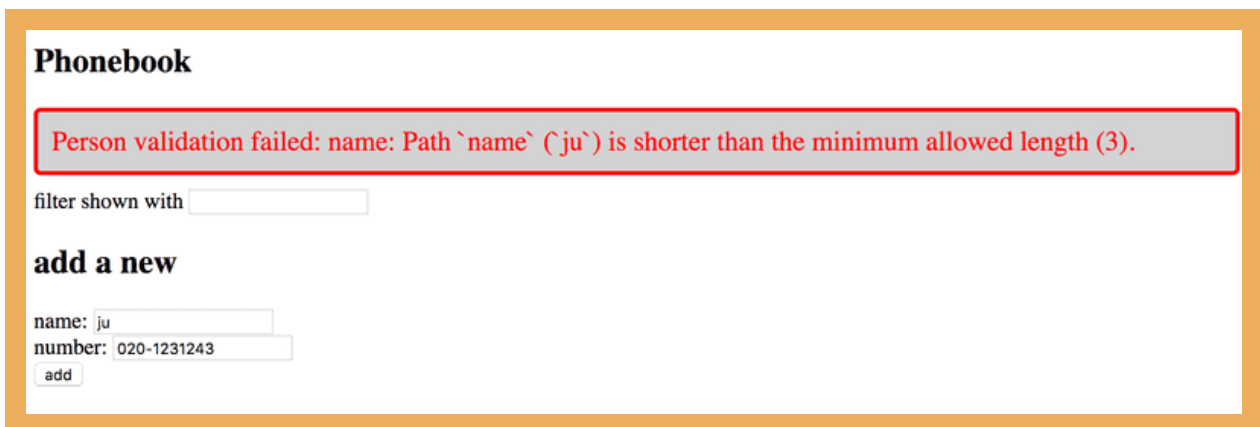
### 3.20\*: Phonebook database, step8

Expand the validation so that the name stored in the database has to be at least three characters long, and the phone number must have at least 8 digits.

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)
  })
```

You can display the default error message returned by Mongoose, even though they are not as readable as they could be:



The screenshot shows a web application titled "Phonebook". At the top, there is a red error message box that reads: "Person validation failed: name: Path `name` (`ju`) is shorter than the minimum allowed length (3)". Below the error message, there is a search bar labeled "filter shown with". Underneath the search bar, there is a section titled "add a new" with two input fields: "name: ju" and "number: 020-1231243". There is an "add" button below the number field.

NB: On update operations, mongoose validators are off by default. [Read the documentation](#) to determine how to enable them.

### 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 copy 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 Heroku and verify that everything works there as well.

## Lint

Before we move onto 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.

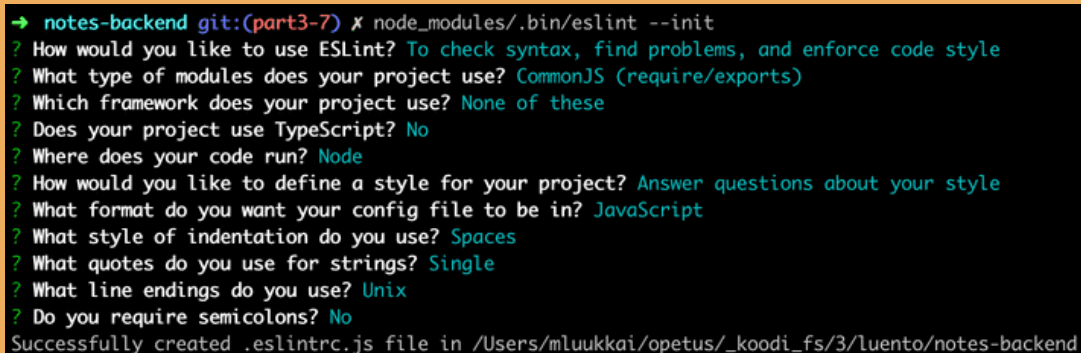
Let's install ESLint as a development dependency to the backend project with the command:

```
npm install eslint --save-dev
```

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

```
node_modules/.bin/eslint --init
```

We will answer all of the questions:

A terminal window with a dark background and orange border. It shows the command 'node\_modules/.bin/eslint --init' being executed. The terminal displays a series of questions and answers for configuring ESLint. The questions are in green, and the answers are in blue. The final message is 'Successfully created .eslintrc.js file in /Users/mluukkai/opetus/\_koodi\_fs/3/luento/notes-backend'.

```
→ notes-backend git:(part3-7) x node_modules/.bin/eslint --init
? How would you like to use ESLint? To check syntax, find problems, and enforce code style
? What type of modules does your project use? CommonJS (require/exports)
? Which framework does your project use? None of these
? Does your project use TypeScript? No
? Where does your code run? Node
? How would you like to define a style for your project? Answer questions about your style
? What format do you want your config file to be in? JavaScript
? What style of indentation do you use? Spaces
? What quotes do you use for strings? Single
? What line endings do you use? Unix
? Do you require semicolons? No
Successfully created .eslintrc.js file in /Users/mluukkai/opetus/_koodi_fs/3/luento/notes-backend
```

The configuration will be saved in the `.eslintrc.js` file:

```
module.exports = {
  'env': {
    'commonjs': true,
    'es2021': true,
    'node': true
  },
  'extends': 'eslint:recommended',
```

```
'parserOptions': {
  'ecmaVersion': 12
},
'rules': {
  'indent': [
    'error',
    4
  ],
  'linebreak-style': [
    'error',
    'unix'
  ],
  'quotes': [
    'error',
    'single'
  ],
  'semi': [
    'error',
    'never'
  ],
  'eqeqeq': 'error',
  'no-trailing-spaces': 'error',
  'object-curly-spacing': [
    'error', 'always'
  ],
  'arrow-spacing': [
    'error', { 'before': true, 'after': true }
  ]
}
```

Let's immediately change the rule concerning indentation, so that the indentation level is two spaces.

```
"indent": [
  "error",
  2
],
```

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

```
node_modules/.bin/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 ."
  },
  // ...
}
```

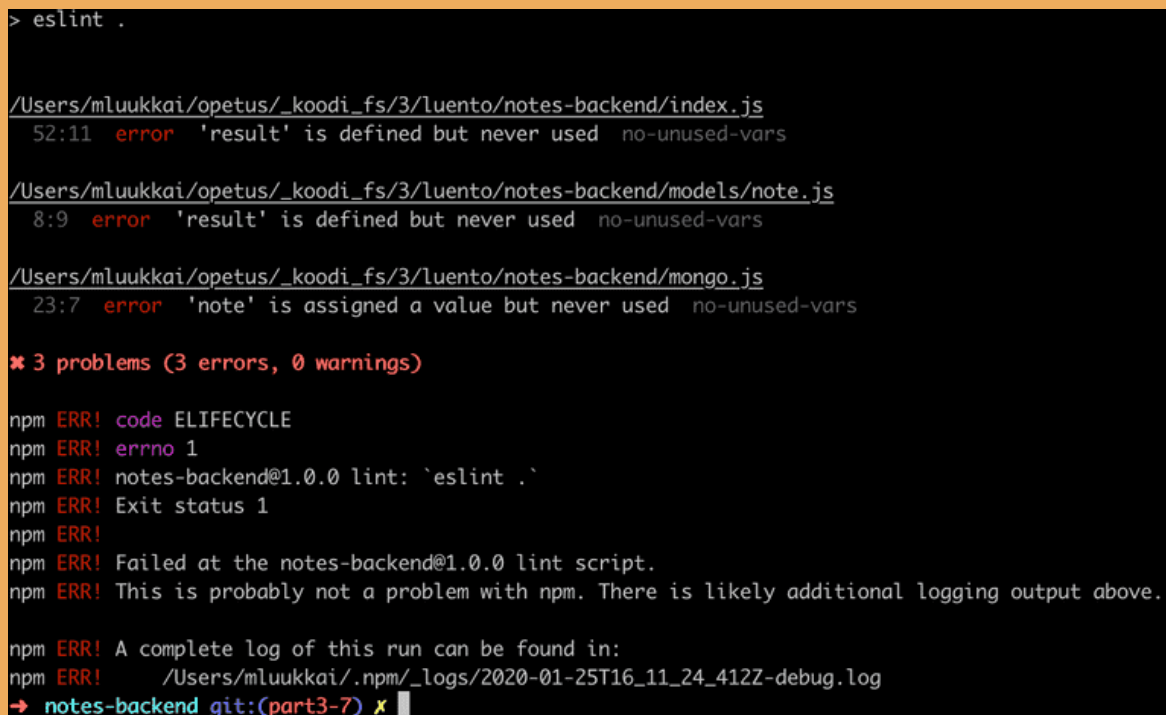
Now the `npm run lint` command will check every file in the project.

Also the files in the `build` directory get checked when the command is run. We do not want this to happen, and we can accomplish this by creating an `.eslintignore` file in the project's root with the following contents:

```
build
```

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

Lint has quite a lot to say about our code:



```
> eslint .

/Users/mluukkai/opetus/_koodi_fs/3/luento/notes-backend/index.js
 52:11  error  'result' is defined but never used  no-unused-vars

/Users/mluukkai/opetus/_koodi_fs/3/luento/notes-backend/models/note.js
   8:9  error  'result' is defined but never used  no-unused-vars

/Users/mluukkai/opetus/_koodi_fs/3/luento/notes-backend/mongo.js
  23:7  error  'note' is assigned a value but never used  no-unused-vars

✖ 3 problems (3 errors, 0 warnings)

npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! notes-backend@1.0.0 lint: `eslint .`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the notes-backend@1.0.0 lint script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.

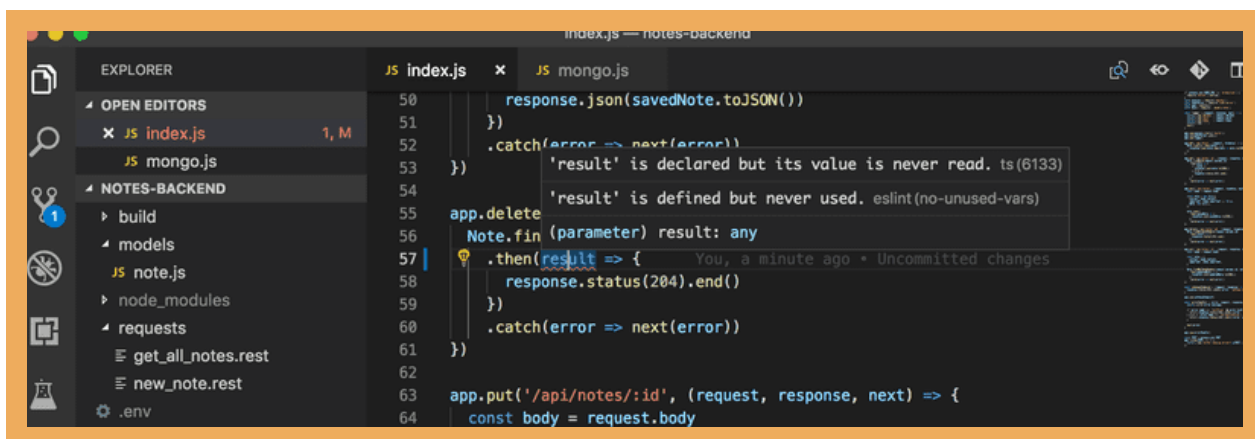
npm ERR! A complete log of this run can be found in:
npm ERR!   /Users/mluukkai/.npm/_logs/2020-01-25T16_11_24_412Z-debug.log
→ notes-backend git:(part3-7) x
```

Let's not fix these issues just yet.

A better alternative to executing the linter from the command line is to configure a *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 `.eslintrc.js` 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.

```
{
  // ...
  '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, let's require that [there is always a space before and after curly braces](#), and let's also demand a consistent use of whitespaces in the function parameters of arrow functions.

```
{
  // ...
  '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 predetermined rules into use from *eslint:recommended*:

```
'extends': 'eslint:recommended',
```

This includes a rule that warns about `console.log` commands. Disabling a rule can be accomplished by defining its "value" as 0 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': 0  
  },  
}
```

NB when you make changes to the *.eslintrc.js* file, it is recommended to run the linter from the command line. This will verify that the configuration file is correctly formatted:

```

→ notes-backend git:(master) X npm run lint

> hello@1.0.0 lint /Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend
> eslint .

/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/.eslintrc.js:30
  "error", {"always"}
                ^
SyntaxError: Unexpected token }
    at new Script (vm.js:74:7)
    at createScript (vm.js:246:10)
    at Object.runInThisContext (vm.js:298:10)
    at Module._compile (internal/modules/cjs/loader.js:646:28)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:689:10)
    at Module.load (internal/modules/cjs/loader.js:589:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:528:12)
    at Function.Module._load (internal/modules/cjs/loader.js:509:3)

```

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

About course

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](#).

FAQ

Partners

[Propose changes to material](#)

Part 3c  
Challenge  
Previous part

Part 4  
Next part

**HOUSTON**