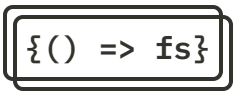


Node.js and Express

In this part, our focus shifts towards the backend: that is, towards implementing functionality on the server side of the stack.

We will be building our backend on top of NodeJS, which is a JavaScript runtime based on Google's Chrome V8 JavaScript engine.

This course material was written with version *v20.11.0* of Node.js. Please make sure that your version of Node is at least as new as the version used in the material (you can check the version by running `node -v` in the command line).

As mentioned in part 1, browsers don't yet support the newest features of JavaScript, and that is why the code running in the browser must be *transpiled* with e.g. babel. The situation with JavaScript running in the backend is different. The newest version of Node supports a large majority of the latest features of JavaScript, so we can use the latest features without having to transpile our code.

Our goal is to implement a backend that will work with the notes application from part 2. However, let's start with the basics by implementing a classic "hello world" application.

Notice that the applications and exercises in this part are not all React applications, and we will not use the *create vite@latest -- --template react* utility for initializing the project for this application.

We had already mentioned npm back in part 2, which is a tool used for managing JavaScript packages. In fact, npm originates from the Node ecosystem.

Let's navigate to an appropriate directory, and create a new template for our application with the `npm init` command. We will answer the questions presented by the utility, and the result will be an automatically generated *package.json* file at the root of the project that contains information about the project.

```
{
  "name": "backend",
  "version": "0.0.1",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Matti Luukkainen",
  "license": "MIT"
}
```

The file defines, for instance, that the entry point of the application is the *index.js* file.

Let's make a small change to the *scripts* object by adding a new script command.

```
{
  // ...
  "scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  // ...
}
```

Next, let's create the first version of our application by adding an *index.js* file to the root of the project with the following code:

```
console.log('hello world')
```

We can run the program directly with Node from the command line:

```
node index.js
```

Or we can run it as an npm script:

```
npm start
```

The *start* npm script works because we defined it in the *package.json* file:

```
{
  // ...
  "scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  // ...
}
```

Even though the execution of the project works when it is started by calling `node index.js` from the command line, it's customary for npm projects to execute such tasks as npm scripts.

By default, the *package.json* file also defines another commonly used npm script called *npm test*. Since our project does not yet have a testing library, the `npm test` command simply executes the following command:

```
echo "Error: no test specified" && exit 1
```

Simple web server

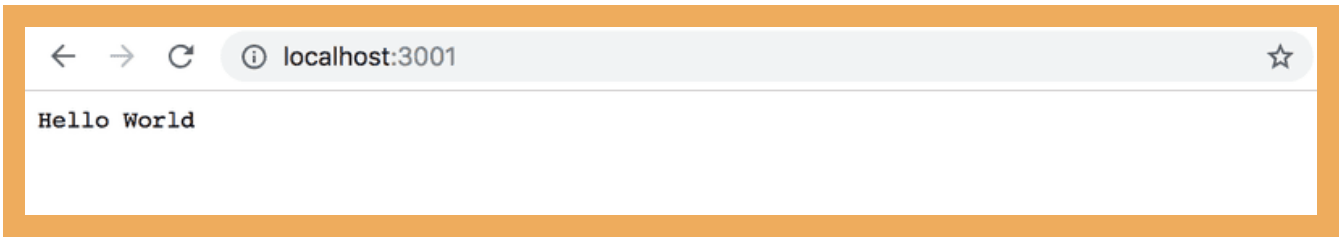
Let's change the application into a web server by editing the `index.js` file as follows:

```
const http = require('http')

const app = http.createServer((request, response) => {
  response.writeHead(200, { 'Content-Type': 'text/plain' })
  response.end('Hello World')
})

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

Once the application is running, the following message is printed in the console:

```
Server running on port 3001
```

We can open our humble application in the browser by visiting the address http://localhost:3001 :



The server works the same way regardless of the latter part of the URL. Also the address http://localhost:3001/foo/bar will display the same content.

NB If port 3001 is already in use by some other application, then starting the server will result in the following error message:

```
→  hello npm start

> hello@1.0.0 start /Users/mluukkai/opetus/_2019fullstack-code/part3/hello
> node index.js

Server running on port 3001
events.js:167
      throw er; // Unhandled 'error' event
      ^

Error: listen EADDRINUSE :::3001
    at Server.setupListenHandle [as _listen2] (net.js:1330:14)
    at listenInCluster (net.js:1378:12)
```

You have two options. Either shut down the application using port 3001 (the JSON Server in the last part of the material was using port 3001), or use a different port for this application.

Let's take a closer look at the first line of the code:

```
const http = require('http')
```

In the first row, the application imports Node's built-in web server module. This is practically what we have already been doing in our browser-side code, but with a slightly different syntax:

```
import http from 'http'
```

These days, code that runs in the browser uses ES6 modules. Modules are defined with an export and included in the current file with an import.

Node.js uses CommonJS modules. The reason for this is that the Node ecosystem needed modules long before JavaScript supported them in the language specification. Currently, Node also supports the use

of ES6 modules, but since the support is not quite perfect yet, we'll stick to CommonJS modules.

CommonJS modules function almost exactly like ES6 modules, at least as far as our needs in this course are concerned.

The next chunk in our code looks like this:

```
const app = http.createServer((request, response) => {
  response.writeHead(200, { 'Content-Type': 'text/plain' })
  response.end('Hello World')
})
```

The code uses the `createServer` method of the `http` module to create a new web server. An *event handler* is registered to the server that is called *every time* an HTTP request is made to the server's address `http://localhost:3001`.

The request is responded to with the status code 200, with the *Content-Type* header set to *text/plain*, and the content of the site to be returned set to *Hello World*.

The last rows bind the http server assigned to the `app` variable, to listen to HTTP requests sent to port 3001:

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

The primary purpose of the backend server in this course is to offer raw data in JSON format to the frontend. For this reason, let's immediately change our server to return a hardcoded list of notes in the JSON format:

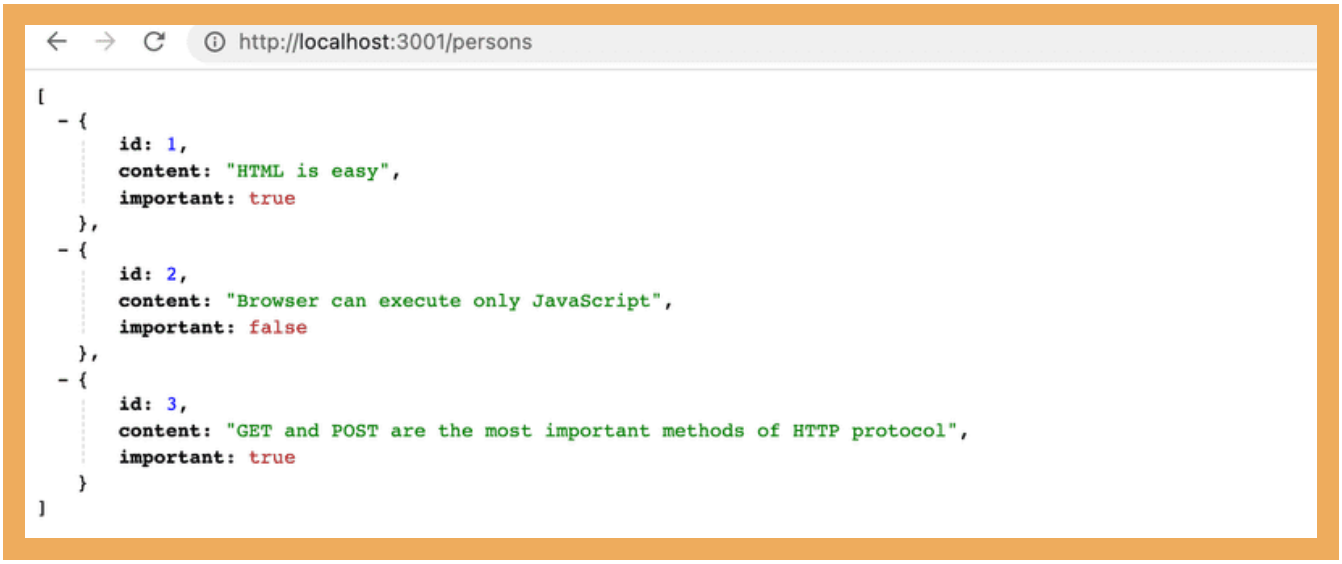
```
const http = require('http')

let notes = [
  {
    id: 1,
    content: "HTML is easy",
    important: true
  },
  {
    id: 2,
    content: "Browser can execute only JavaScript",
    important: false
  },
  {
    id: 3,
    content: "GET and POST are the most important methods of HTTP protocol",
```

```
    important: true
  }
]
const app = http.createServer((request, response) => {
  response.writeHead(200, { 'Content-Type': 'application/json' })
  response.end(JSON.stringify(notes))
})

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

Let's restart the server (you can shut the server down by pressing `Ctrl+C` in the console) and let's refresh the browser.

The *application/json* value in the *Content-Type* header informs the receiver that the data is in the JSON format. The `notes` array gets transformed into JSON formatted string with the `JSON.stringify(notes)` method. This is necessary because the response.end() method expects a string or a buffer to send as the response body.

When we open the browser, the displayed format is exactly the same as in part 2 where we used json-server to serve the list of notes:



Express

Implementing our server code directly with Node's built-in http web server is possible. However, it is cumbersome, especially once the application grows in size.

Many libraries have been developed to ease server-side development with Node, by offering a more pleasing interface to work with the built-in http module. These libraries aim to provide a better abstraction for general use cases we usually require to build a backend server. By far the most popular library intended for this purpose is Express.

Let's take Express into use by defining it as a project dependency with the command:

```
npm install express
```

copy

The dependency is also added to our *package.json* file:

```
{
  // ...
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

copy

The source code for the dependency is installed in the *node_modules* directory located at the root of the project. In addition to Express, you can find a great number of other dependencies in the directory:

These are the dependencies of the Express library and the dependencies of all of its dependencies, and so forth. These are called the transitive dependencies of our project.

Version 4.18.2 of Express was installed in our project. What does the caret in front of the version number in *package.json* mean?

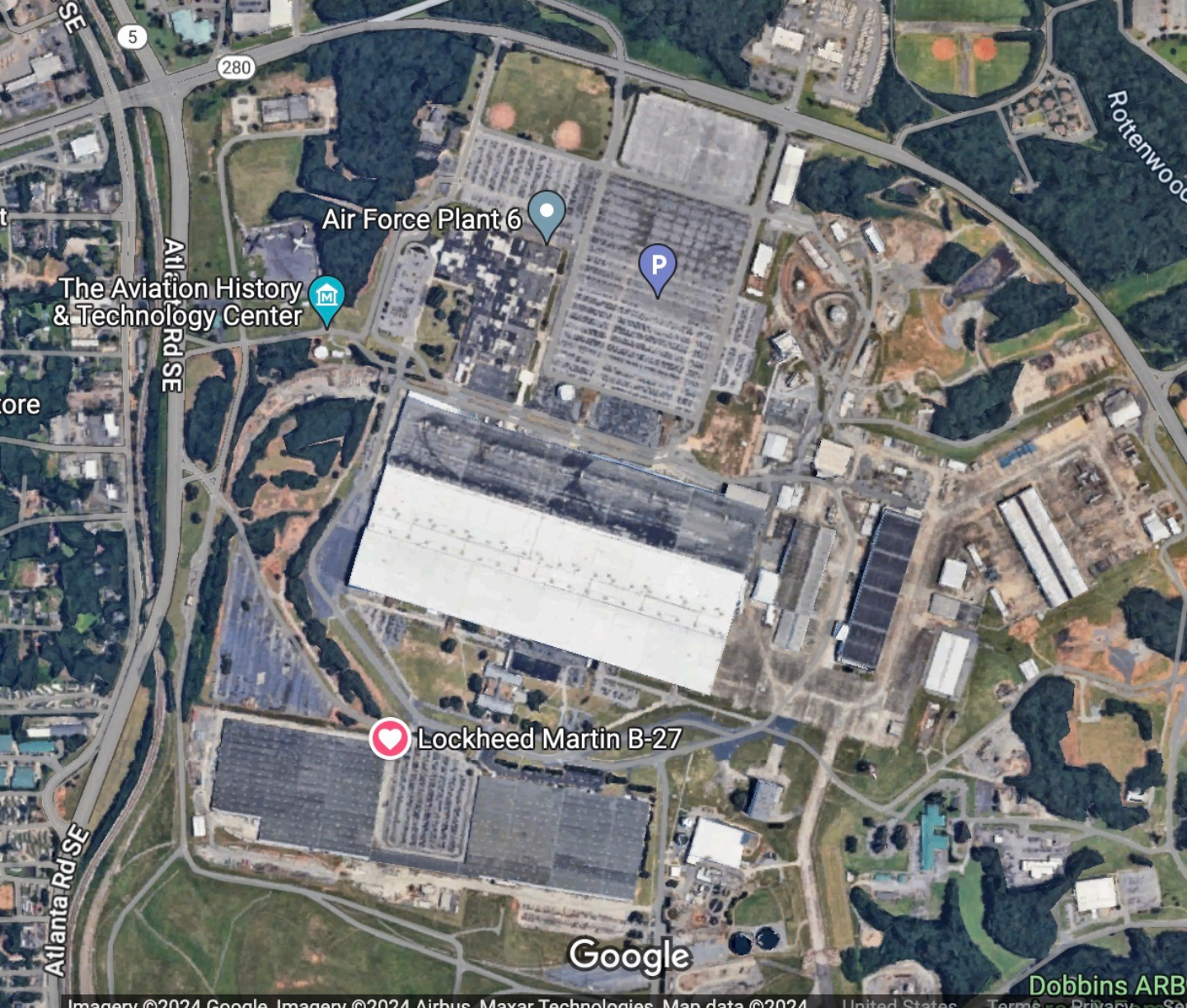
```
"express": "^4.18.2"
```

copy

The versioning model used in npm is called semantic versioning.

The caret in the front of *^4.18.2* means that if and when the dependencies of a project are updated, the version of Express that is installed will be at least *4.18.2*. However, the installed version of Express can also have a larger *patch* number (the last number), or a larger *minor* number (the middle number). The major version of the library indicated by the first *major* number must be the same.

We can update the dependencies of the project with the command:


5
280
Air Force Plant 6
P
The Aviation History
& Technology Center
Atlanta Rd SE
Rottenwood
Lockheed Martin B-27
Google
Dobbins ARB

```
npm update
```

Likewise, if we start working on the project on another computer, we can install all up-to-date dependencies of the project defined in *package.json* by running this next command in the project's root directory:

```
npm install
```

If the *major* number of a dependency does not change, then the newer versions should be backwards compatible. This means that if our application happened to use version 4.99.175 of Express in the future, then all the code implemented in this part would still have to work without making changes to the code. In contrast, the future 5.0.0 version of Express may contain changes that would cause our application to no longer work.

Web and Express

Let's get back to our application and make the following changes:

```
const express = require('express')
const app = express()

let notes = [
  ...
]

app.get('/', (request, response) => {
  response.send('<h1>Hello World!</h1>')
})

app.get('/api/notes', (request, response) => {
  response.json(notes)
})

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

To get the new version of our application into use, first we have to restart it.

The application did not change a whole lot. Right at the beginning of our code, we're importing `express`, which this time is a *function* that is used to create an Express application stored in the `app` variable:

```
const express = require('express')
const app = express()
```

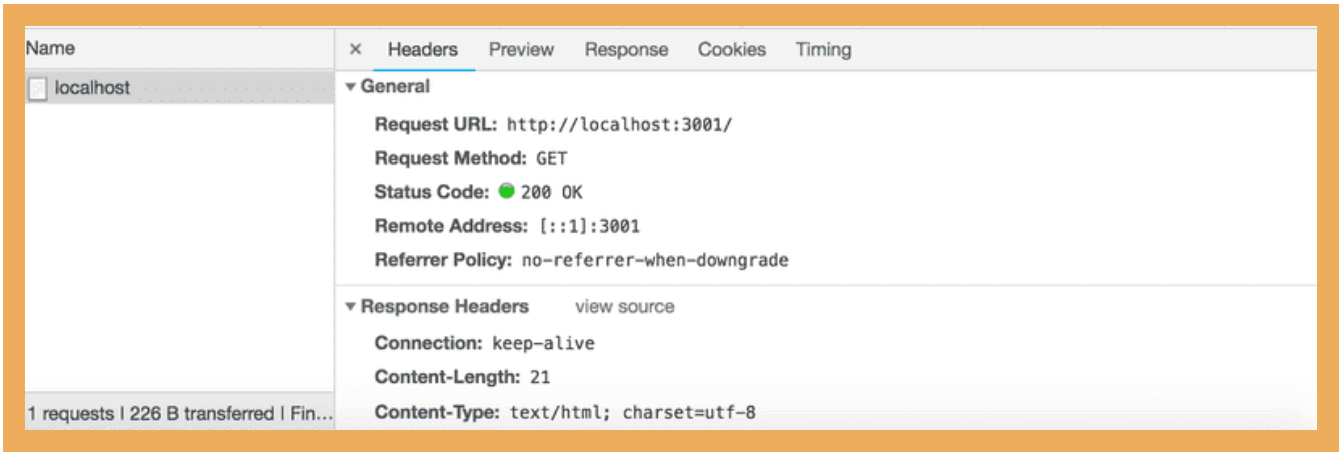
Next, we define two *routes* to the application. The first one defines an event handler that is used to handle HTTP GET requests made to the application's / root:

```
app.get('/', (request, response) => {
  response.send('<h1>Hello World!</h1>')
})
```

The event handler function accepts two parameters. The first `request` parameter contains all of the information of the HTTP request, and the second `response` parameter is used to define how the request is responded to.

In our code, the request is answered by using the `send` method of the `response` object. Calling the method makes the server respond to the HTTP request by sending a response containing the string `<h1>Hello World!</h1>` that was passed to the `send` method. Since the parameter is a string, Express automatically sets the value of the *Content-Type* header to be *text/html*. The status code of the response defaults to 200.

We can verify this from the *Network* tab in developer tools:

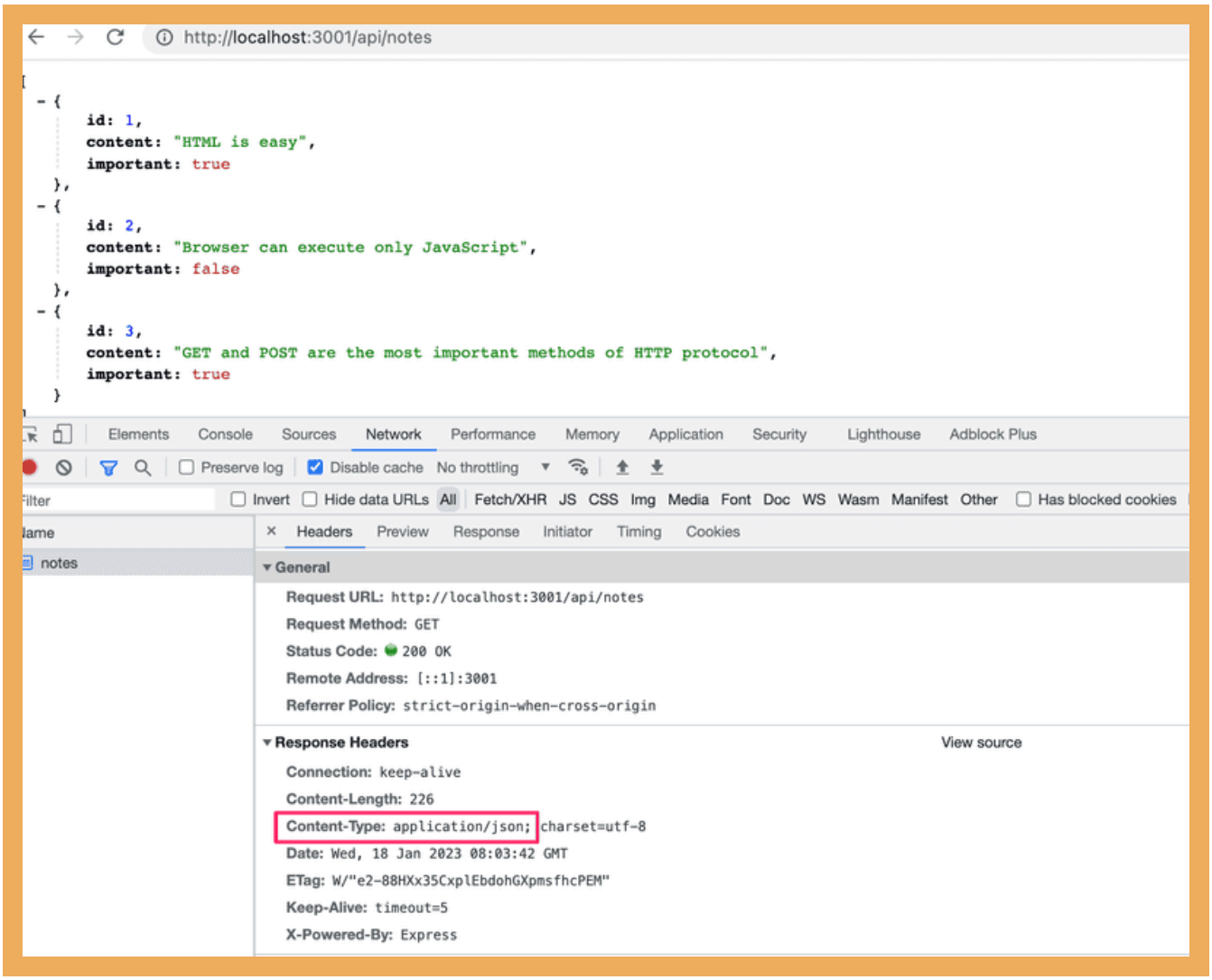


The second route defines an event handler that handles HTTP GET requests made to the *notes* path of the application:

```
app.get('/api/notes', (request, response) => {
  response.json(notes)
})
```

The request is responded to with the `json` method of the `response` object. Calling the method will send the **notes** array that was passed to it as a JSON formatted string. Express automatically sets the

Content-Type header with the appropriate value of *application/json*.



Next, let's take a quick look at the data sent in JSON format.

In the earlier version where we were only using Node, we had to transform the data into the JSON formatted string with the `JSON.stringify` method:

```
response.end(JSON.stringify(notes))
```

copy

With Express, this is no longer required, because this transformation happens automatically.

It's worth noting that `JSON` is a string and not a JavaScript object like the value assigned to `notes`.

The experiment shown below illustrates this point:

The experiment above was done in the interactive node-repl. You can start the interactive node-repl by typing in `node` in the command line. The repl is particularly useful for testing how commands work while you're writing application code. I highly recommend this!

nodemon

If we make changes to the application's code we have to restart the application to see the changes. We restart the application by first shutting it down by typing `Ctrl+C` and then restarting the application. Compared to the convenient workflow in React where the browser automatically reloaded after changes were made, this feels slightly cumbersome.

The solution to this problem is nodemon:

> *nodemon will watch the files in the directory in which nodemon was started, and if any files change, nodemon will automatically restart your node application.*

Let's install nodemon by defining it as a *development dependency* with the command:

```
npm install --save-dev nodemon
```

copy

The contents of *package.json* have also changed:

```
{
  //...
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "nodemon": "^3.0.3"
```

copy

```
  }
}
```

If you accidentally used the wrong command and the nodemon dependency was added under "dependencies" instead of "devDependencies", then manually change the contents of *package.json* to match what is shown above.

By development dependencies, we are referring to tools that are needed only during the development of the application, e.g. for testing or automatically restarting the application, like *nodemon*.

These development dependencies are not needed when the application is run in production mode on the production server (e.g. Fly.io or Heroku).

We can start our application with *nodemon* like this:

```
node_modules/.bin/nodemon index.js
```

copy

Changes to the application code now cause the server to restart automatically. It's worth noting that even though the backend server restarts automatically, the browser still has to be manually refreshed. This is because unlike when working in React, we don't have the hot reload functionality needed to automatically reload the browser.

The command is long and quite unpleasant, so let's define a dedicated *npm script* for it in the *package.json* file:

```
{
  // ..
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  // ..
}
```

copy

In the script there is no need to specify the *node_modules/.bin/nodemon* path to nodemon, because `npm` automatically knows to search for the file from that directory.

We can now start the server in development mode with the command:

```
npm run dev
```

copy

Unlike with the *start* and *test* scripts, we also have to add *run* to the command because it is a non-native script.

REST

Let's expand our application so that it provides the same RESTful HTTP API as json-server.

Representational State Transfer, aka REST, was introduced in 2000 in Roy Fielding's dissertation. REST is an architectural style meant for building scalable web applications.

We are not going to dig into Fielding's definition of REST or spend time pondering about what is and isn't RESTful. Instead, we take a more narrow view by only concerning ourselves with how RESTful APIs are typically understood in web applications. The original definition of REST is not even limited to web applications.

We mentioned in the previous part that singular things, like notes in the case of our application, are called *resources* in RESTful thinking. Every resource has an associated URL which is the resource's unique address.

One convention for creating unique addresses is to combine the name of the resource type with the resource's unique identifier.

Let's assume that the root URL of our service is *www.example.com/api*.

If we define the resource type of note to be *notes*, then the address of a note resource with the identifier 10, has the unique address *www.example.com/api/notes/10*.

The URL for the entire collection of all note resources is *www.example.com/api/notes*.

We can execute different operations on resources. The operation to be executed is defined by the HTTP *verb*:

URL	verb	functionality
notes/10	GET	fetches a single resource
notes	GET	fetches all resources in the collection
notes	POST	creates a new resource based on the request data
notes/10	DELETE	removes the identified resource
notes/10	PUT	replaces the entire identified resource with the request data
notes/10	PATCH	replaces a part of the identified resource with the request data

This is how we manage to roughly define what REST refers to as a uniform interface, which means a consistent way of defining interfaces that makes it possible for systems to cooperate.

This way of interpreting REST falls under the second level of RESTful maturity in the Richardson Maturity Model. According to the definition provided by Roy Fielding, we have not defined a REST API. In fact, a large majority of the world's purported "REST" APIs do not meet Fielding's original criteria outlined in his dissertation.

In some places (see e.g. Richardson, Ruby: RESTful Web Services) you will see our model for a straightforward CRUD API, being referred to as an example of resource-oriented architecture instead of REST. We will avoid getting stuck arguing semantics and instead return to working on our application.

Fetching a single resource

Let's expand our application so that it offers a REST interface for operating on individual notes. First, let's create a route for fetching a single resource.

The unique address we will use for an individual note is of the form *notes/10*, where the number at the end refers to the note's unique id number.

We can define parameters for routes in Express by using the colon syntax:

```
app.get('/api/notes/:id', (request, response) => {
  const id = request.params.id
  const note = notes.find(note => note.id === id)
  response.json(note)
})
```

Now `app.get('/api/notes/:id', ...)` will handle all HTTP GET requests that are of the form */api/notes/SOMETHING*, where *SOMETHING* is an arbitrary string.

The *id* parameter in the route of a request can be accessed through the request object:

```
const id = request.params.id
```

The now familiar `find` method of arrays is used to find the note with an id that matches the parameter. The note is then returned to the sender of the request.

When we test our application by going to http://localhost:3001/api/notes/1 in our browser, we notice that it does not appear to work, as the browser displays an empty page. This comes as no surprise to us as software developers, and it's time to debug.

Adding `console.log` commands into our code is a time-proven trick:

```
app.get('/api/notes/:id', (request, response) => {
  const id = request.params.id
  console.log(id)
```

```
  const note = notes.find(note => note.id === id)
  console.log(note)
  response.json(note)
})
```

When we visit http://localhost:3001/api/notes/1 again in the browser, the console - which is the terminal (in this case) - will display the following:

```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Server running on port 3001
1
undefined
```

The id parameter from the route is passed to our application but the `find` method does not find a matching note.

To further our investigation, we also add a console log inside the comparison function passed to the `find` method. To do this, we have to get rid of the compact arrow function syntax `note => note.id === id` , and use the syntax with an explicit return statement:

```
app.get('/api/notes/:id', (request, response) => {
  const id = request.params.id
  const note = notes.find(note => {
    console.log(note.id, typeof note.id, id, typeof id, note.id === id)
    return note.id === id
  })
  console.log(note)
  response.json(note)
})
```

copy

When we visit the URL again in the browser, each call to the comparison function prints a few different things to the console. The console output is the following:

```
1 'number' '1' 'string' false
2 'number' '1' 'string' false
3 'number' '1' 'string' false
```

copy

The cause of the bug becomes clear. The `id` variable contains a string '1', whereas the ids of notes are integers. In JavaScript, the "triple equals" comparison === considers all values of different types to not be equal by default, meaning that 1 is not '1'.

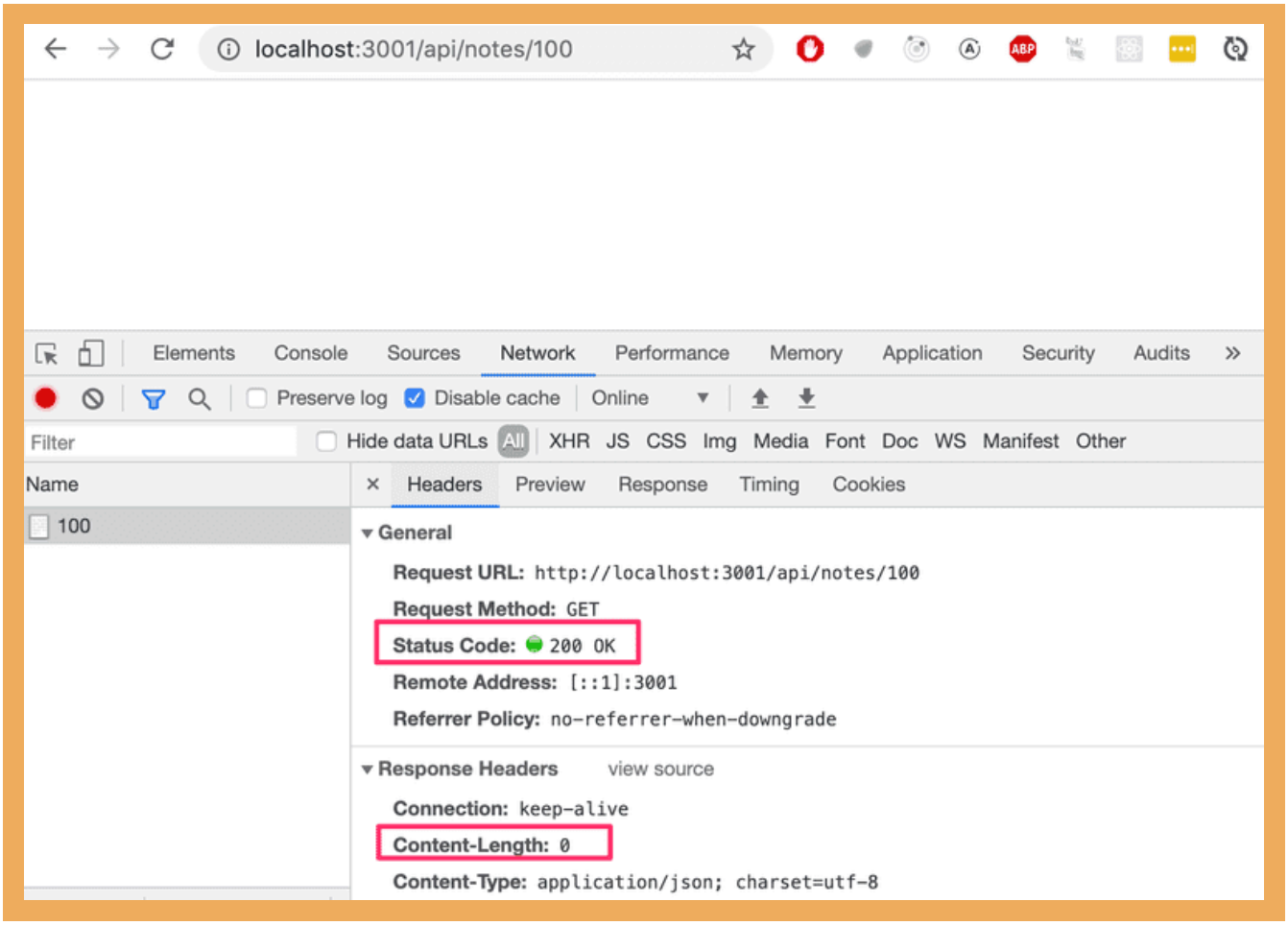
Let's fix the issue by changing the id parameter from a string into a number :

```
app.get('/api/notes/:id', (request, response) => {
  const id = Number(request.params.id)
  const note = notes.find(note => note.id === id)
  response.json(note)
})
```

Now fetching an individual resource works.

However, there's another problem with our application.

If we search for a note with an id that does not exist, the server responds with:



The HTTP status code that is returned is 200, which means that the response succeeded. There is no data sent back with the response, since the value of the *content-length* header is 0, and the same can be verified from the browser.

The reason for this behavior is that the `note` variable is set to `undefined` if no matching note is found. The situation needs to be handled on the server in a better way. If no note is found, the server should respond with the status code 404 not found instead of 200.

Let's make the following change to our code:

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

  if (note) {
    response.json(note)
  } else {
    response.status(404).end()
  }
})
```

Since no data is attached to the response, we use the `status` method for setting the status and the `end` method for responding to the request without sending any data.

The if-condition leverages the fact that all JavaScript objects are truthy, meaning that they evaluate to true in a comparison operation. However, `undefined` is falsy meaning that it will evaluate to false.

Our application works and sends the error status code if no note is found. However, the application doesn't return anything to show to the user, like web applications normally do when we visit a page that does not exist. We do not need to display anything in the browser because REST APIs are interfaces that are intended for programmatic use, and the error status code is all that is needed.

Anyway, it's possible to give a clue about the reason for sending a 404 error by overriding the default NOT FOUND message.

Deleting resources

Next, let's implement a route for deleting resources. Deletion happens by making an HTTP DELETE request to the URL of the resource:

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

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

If deleting the resource is successful, meaning that the note exists and is removed, we respond to the request with the status code 204 no content and return no data with the response.

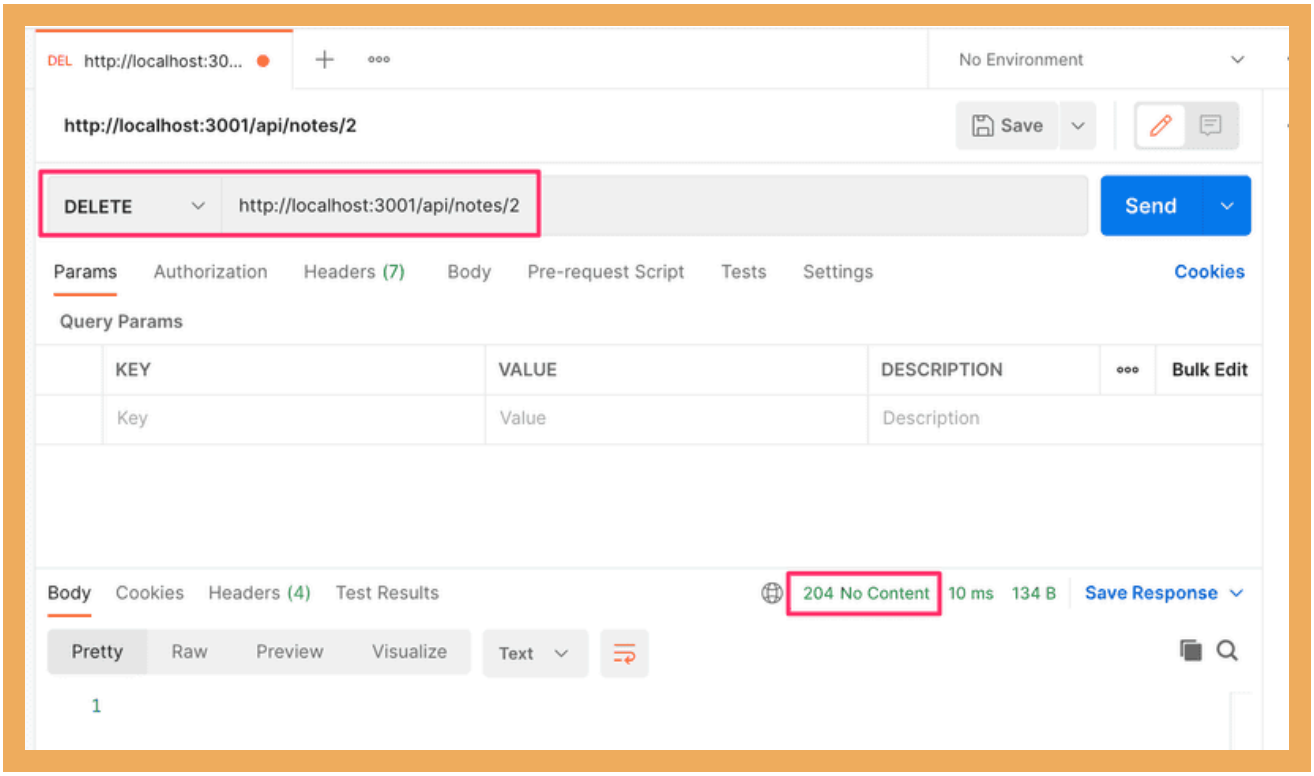
There's no consensus on what status code should be returned to a DELETE request if the resource does not exist. The only two options are 204 and 404. For the sake of simplicity, our application will respond with 204 in both cases.

Postman

So how do we test the delete operation? HTTP GET requests are easy to make from the browser. We could write some JavaScript for testing deletion, but writing test code is not always the best solution in every situation.

Many tools exist for making the testing of backends easier. One of these is a command line program curl. However, instead of curl, we will take a look at using Postman for testing the application.

Let's install the Postman desktop client from here and try it out:



NB: Postman is also available on VS Code which can be downloaded from the Extension tab on the left -> search for Postman -> First result (Verified Publisher) -> Install You will then see an extra icon added on the activity bar below the extensions tab. Once you log in, you can follow the steps below

Using Postman is quite easy in this situation. It's enough to define the URL and then select the correct request type (DELETE).

The backend server appears to respond correctly. By making an HTTP GET request to http://localhost:3001/api/notes we see that the note with the id 2 is no longer in the list, which indicates that the deletion was successful.

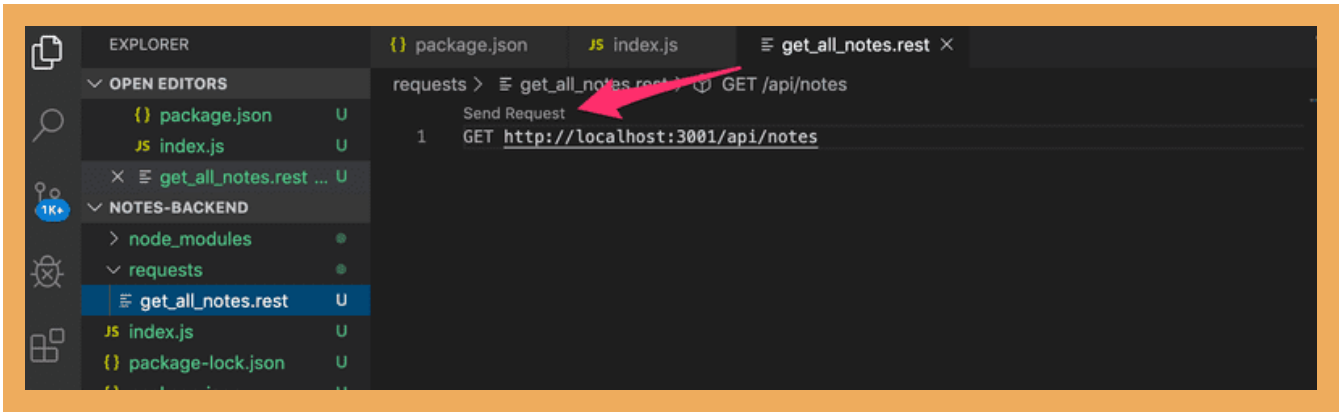
Because the notes in the application are only saved to memory, the list of notes will return to its original state when we restart the application.

The Visual Studio Code REST client

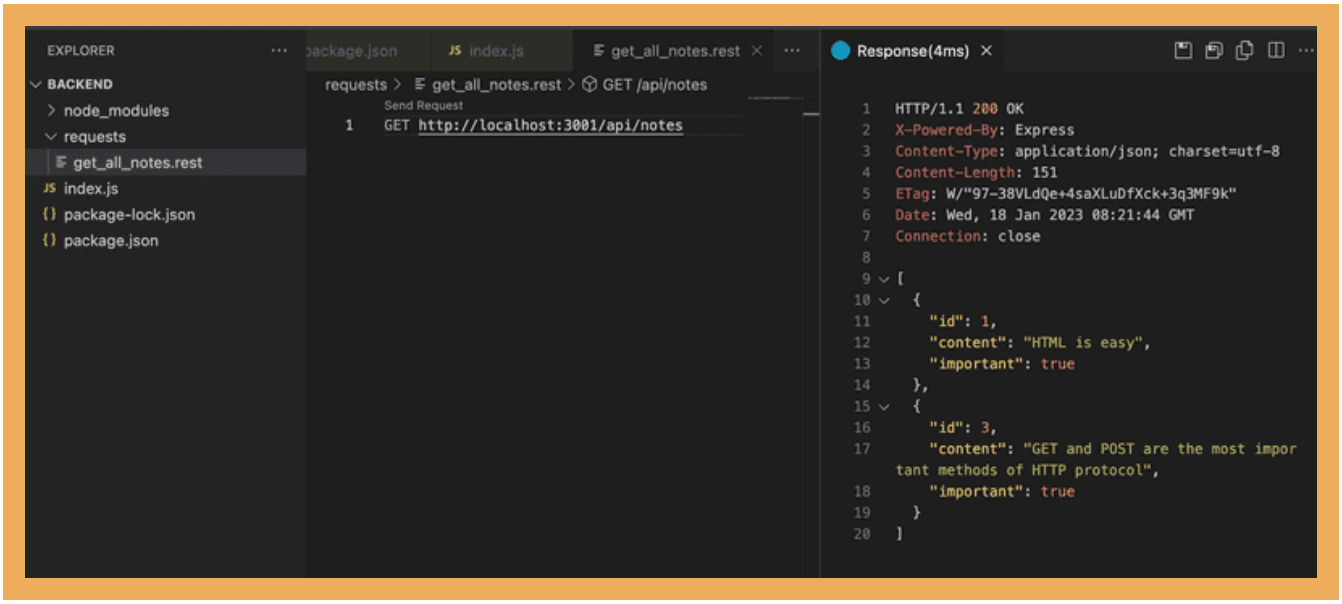
If you use Visual Studio Code, you can use the VS Code REST client plugin instead of Postman.

Once the plugin is installed, using it is very simple. We make a directory at the root of the application named *requests*. We save all the REST client requests in the directory as files that end with the *.rest* extension.

Let's create a new *get_all_notes.rest* file and define the request that fetches all notes.



By clicking the *Send Request* text, the REST client will execute the HTTP request and the response from the server is opened in the editor.



The WebStorm HTTP Client

If you use `IntelliJ WebStorm` instead, you can use a similar procedure with its built-in HTTP Client. Create a new file with extension `.rest` and the editor will display your options to create and run your requests. You can learn more about it by following this guide.

Receiving data

Next, let's make it possible to add new notes to the server. Adding a note happens by making an HTTP POST request to the address http://localhost:3001/api/notes, and by sending all the information for the new note in the request body in JSON format.

To access the data easily, we need the help of the Express json-parser that we can use with the command `app.use(express.json())`.

Let's activate the json-parser and implement an initial handler for dealing with the HTTP POST requests:

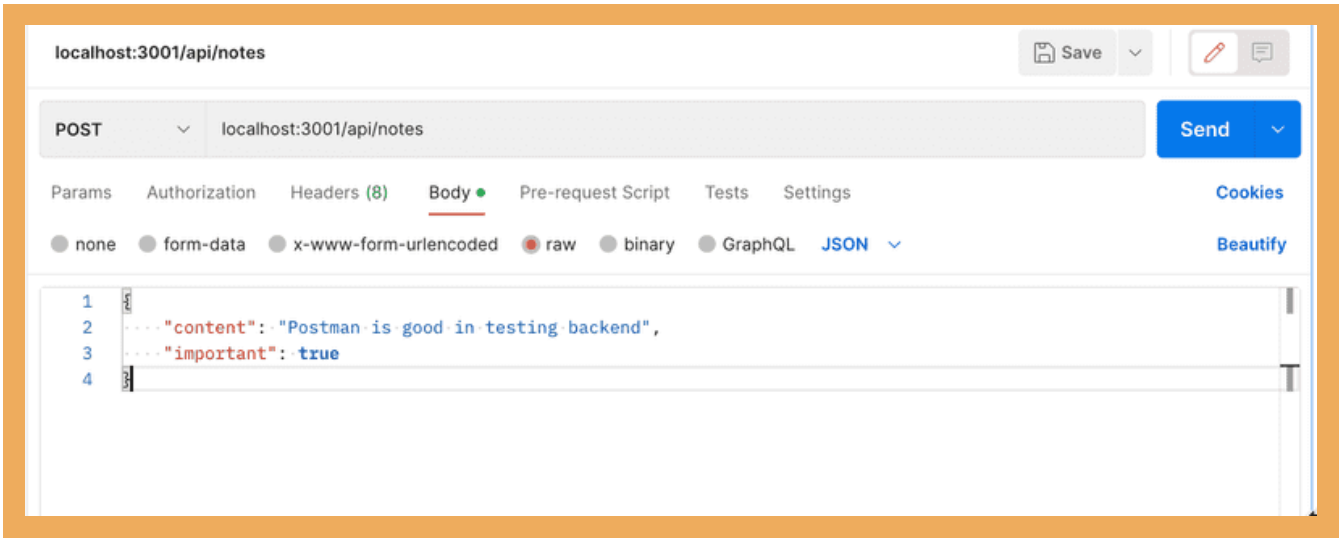
```
const express = require('express')
const app = express()

app.use(express.json())

//...

app.post('/api/notes', (request, response) => {
  const note = request.body
  console.log(note)
  response.json(note)
})
```

The event handler function can access the data from the *body* property of the `request` object.

Without the json-parser, the *body* property would be undefined. The json-parser takes the JSON data of a request, transforms it into a JavaScript object and then attaches it to the *body* property of the `request` object before the route handler is called.

For the time being, the application does not do anything with the received data besides printing it to the console and sending it back in the response.

Before we implement the rest of the application logic, let's verify with Postman that the data is in fact received by the server. In addition to defining the URL and request type in Postman, we also have to define the data sent in the *body*:



The application prints the data that we sent in the request to the console:

```
[nodemon] starting `node index.js`
Server running on port 3001
{ content: 'Postman is good in testing backend', important: true }
```

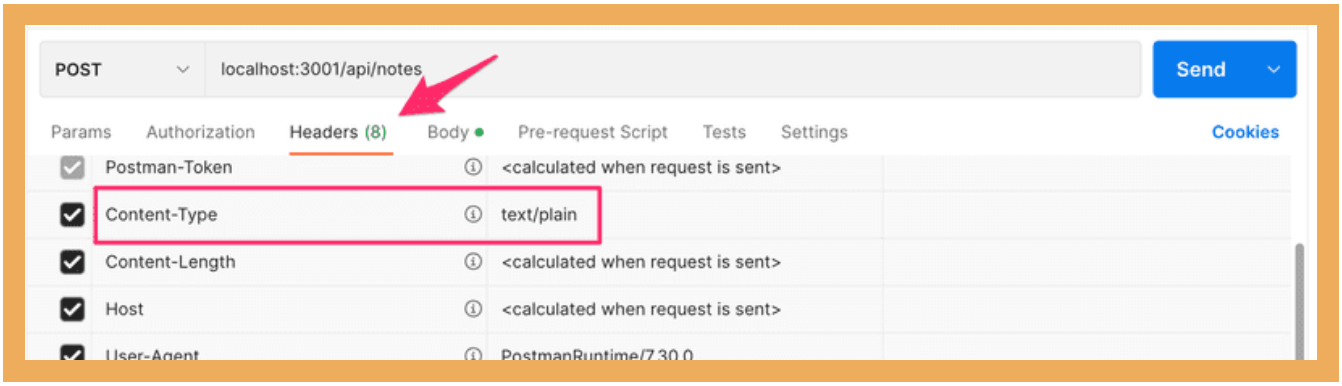
NB *Keep the terminal running the application visible at all times* when you are working on the backend. Thanks to Nodemon any changes we make to the code will restart the application. If you pay attention to the console, you will immediately be able to pick up on errors that occur in the application:

```
[nodemon] starting `node index.js`
/Users/mluukkai/opetus/2024-fs/part3/notes-backend/index.js:22
app.use(express.json()
                  ^

SyntaxError: missing ) after argument list
    at internalCompileFunction (node:internal/vm:74:18)
    at wrapSafe (node:internal/modules/cjs/loader:1141:20)
    at Module._compile (node:internal/modules/cjs/loader:1182:27)
    at Module._extensions..js (node:internal/modules/cjs/loader:1272:10)
    at Module.load (node:internal/modules/cjs/loader:1081:32)
    at Module._load (node:internal/modules/cjs/loader:922:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:23:47
```

Similarly, it is useful to check the console to make sure that the backend behaves as we expect it to in different situations, like when we send data with an HTTP POST request. Naturally, it's a good idea to add lots of `console.log` commands to the code while the application is still being developed.

A potential cause for issues is an incorrectly set *Content-Type* header in requests. This can happen with Postman if the type of body is not defined correctly:

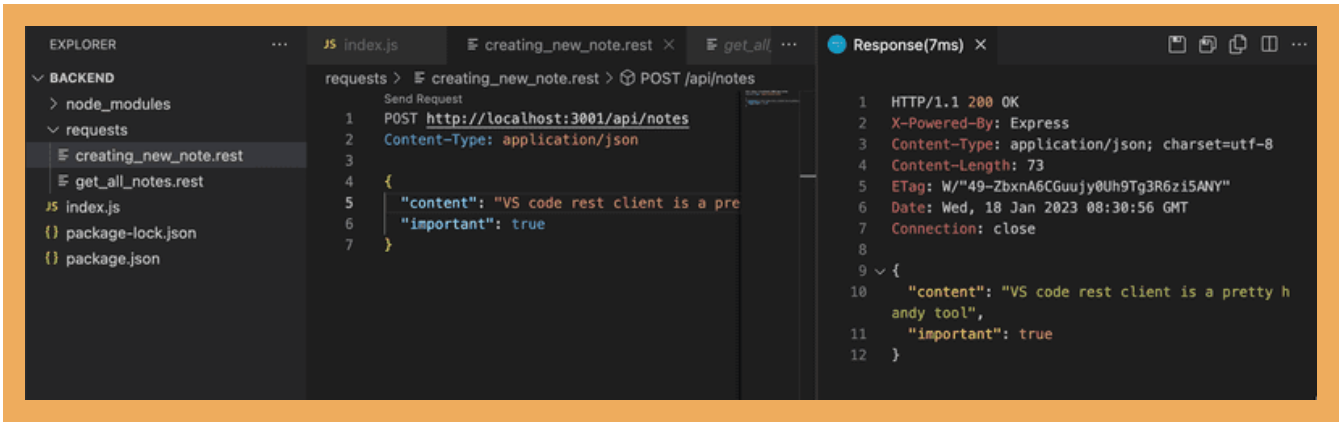
The *Content-Type* header is set to *text/plain*:



The server appears to only receive an empty object:

The server will not be able to parse the data correctly without the correct value in the header. It won't even try to guess the format of the data since there's a massive amount of potential *Content-Types*.

If you are using VS Code, then you should install the REST client from the previous chapter *now, if you haven't already*. The POST request can be sent with the REST client like this:



We created a new *create_note.rest* file for the request. The request is formatted according to the instructions in the documentation.

One benefit that the REST client has over Postman is that the requests are handily available at the root of the project repository, and they can be distributed to everyone in the development team. You can also add multiple requests in the same file using `###` separators:

```
GET http://localhost:3001/api/notes/

###
POST http://localhost:3001/api/notes/ HTTP/1.1
content-type: application/json

{
    "name": "sample",
    "time": "Wed, 21 Oct 2015 18:27:50 GMT"
}
```

Postman also allows users to save requests, but the situation can get quite chaotic especially when you're working on multiple unrelated projects.

> **Important sidenote**
>
> Sometimes when you're debugging, you may want to find out what headers have been set in the HTTP request. One way of accomplishing this is through the `get` method of the `request` object, that can be used for getting the value of a single header. The `request` object also has the *headers* property, that contains all of the headers of a specific request.
>
> Problems can occur with the VS REST client if you accidentally add an empty line between the top row and the row specifying the HTTP headers. In this situation, the REST client interprets this to mean that all headers are left empty, which leads to the backend server not knowing that the data it has received is in the JSON format.
>
> You will be able to spot this missing *Content-Type* header if at some point in your code you print all of the request headers with the `console.log(request.headers)` command.

Let's return to the application. Once we know that the application receives data correctly, it's time to finalize the handling of the request:

```
app.post('/api/notes', (request, response) => {
  const maxId = notes.length > 0
    ? Math.max(...notes.map(n => n.id))
    : 0

  const note = request.body
  note.id = maxId + 1

  notes = notes.concat(note)
```

```
  response.json(note)
})
```

We need a unique id for the note. First, we find out the largest id number in the current list and assign it to the `maxId` variable. The id of the new note is then defined as `maxId + 1`. This method is not recommended, but we will live with it for now as we will replace it soon enough.

The current version still has the problem that the HTTP POST request can be used to add objects with arbitrary properties. Let's improve the application by defining that the *content* property may not be empty. The *important* property will be given default value false. All other properties are discarded:

```
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) {
    return response.status(400).json({
      error: 'content missing'
    })
  }

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

  notes = notes.concat(note)

  response.json(note)
})
```

The logic for generating the new id number for notes has been extracted into a separate `generateId` function.

If the received data is missing a value for the *content* property, the server will respond to the request with the status code 400 bad request:

```
if (!body.content) {
  return response.status(400).json({
    error: 'content missing'
```

```
  })
}
```

Notice that calling return is crucial because otherwise the code will execute to the very end and the malformed note gets saved to the application.

If the content property has a value, the note will be based on the received data. If the *important* property is missing, we will default the value to *false*. The default value is currently generated in a rather odd-looking way:

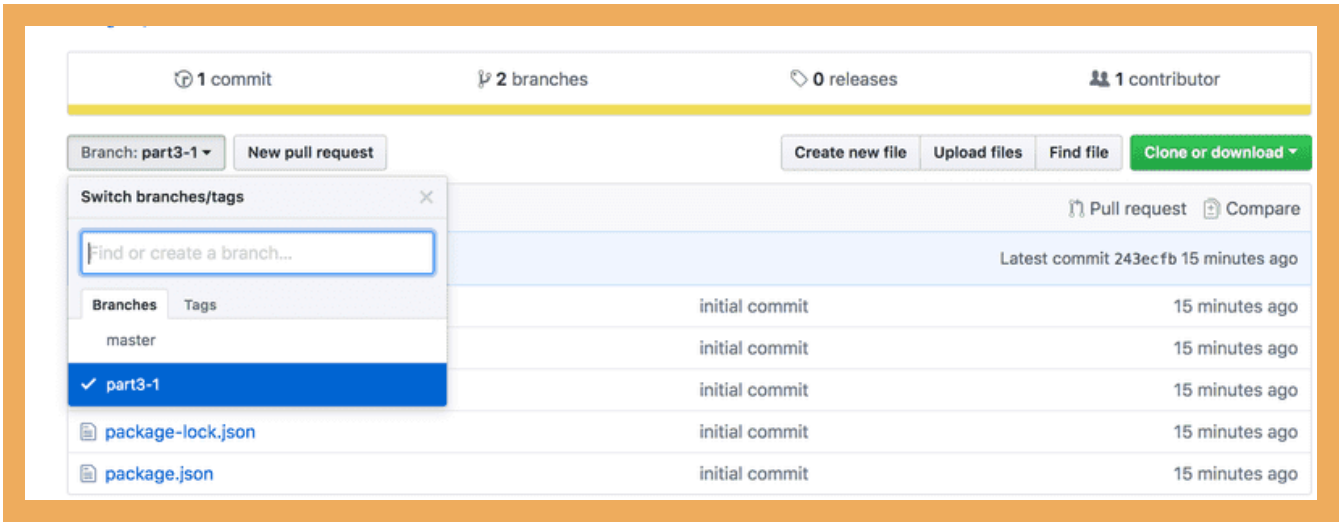
```
important: Boolean(body.important) || false,
```

copy

If the data saved in the `body` variable has the *important* property, the expression will evaluate its value and convert it to a boolean value. If the property does not exist, then the expression will evaluate to false which is defined on the right-hand side of the vertical lines.

> To be exact, when the *important* property is *false*, then the `body.important || false` expression will in fact return the *false* from the right-hand side...

You can find the code for our current application in its entirety in the *part3-1* branch of this GitHub repository.



If you clone the project, run the `npm install` command before starting the application with `npm start` or `npm run dev`.

One more thing before we move on to the exercises. The function for generating IDs looks currently like this:

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

copy

```
  return maxId + 1
}
```

The function body contains a row that looks a bit intriguing:

```
Math.max(...notes.map(n => n.id))
```

copy

What exactly is happening in that line of code? `notes.map(n => n.id)` creates a new array that contains all the ids of the notes. Math.max returns the maximum value of the numbers that are passed to it. However, `notes.map(n => n.id)` is an *array* so it can't directly be given as a parameter to `Math.max`. The array can be transformed into individual numbers by using the "three dot" spread syntax `...`.

Exercises 3.1.-3.6.

NB: It's recommended to do all of the exercises from this part into a new dedicated git repository, and place your source code right at the root of the repository. Otherwise, you will run into problems in exercise 3.10.

NB: Because this is not a frontend project and we are not working with React, the application **is not created** with create vite@latest -- --template react. You initialize this project with the `npm init` command that was demonstrated earlier in this part of the material.

Strong recommendation: When you are working on backend code, always keep an eye on what's going on in the terminal that is running your application.

3.1: Phonebook backend step 1

Implement a Node application that returns a hardcoded list of phonebook entries from the address http://localhost:3001/api/persons.

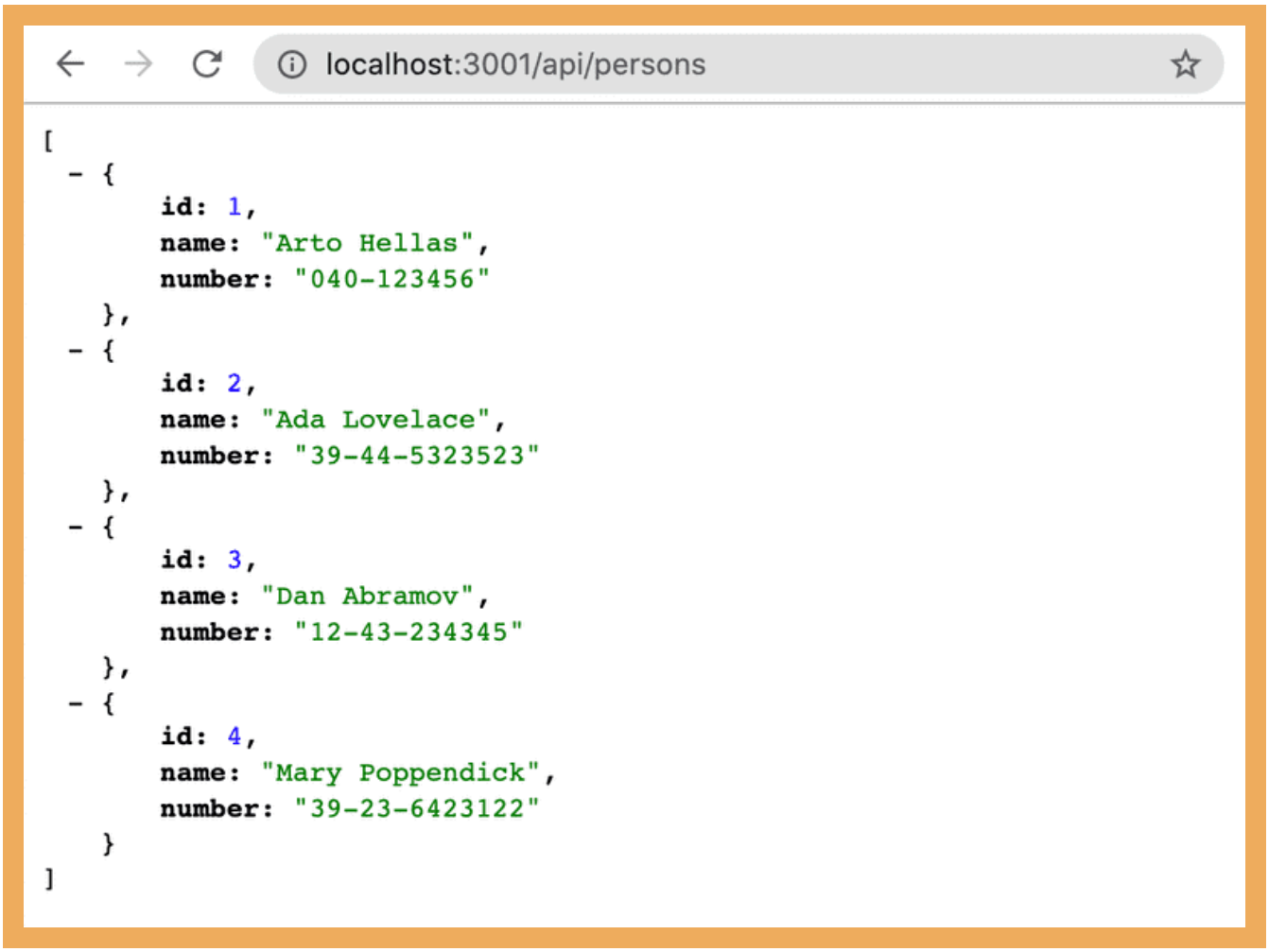
Data:

```
[
    {
      "id": 1,
      "name": "Arto Hellas",
      "number": "040-123456"
    },
    {
      "id": 2,
      "name": "Ada Lovelace",
      "number": "39-44-5323523"
    },
```

copy

```
    {
      "id": 3,
      "name": "Dan Abramov",
      "number": "12-43-234345"
    },
    {
      "id": 4,
      "name": "Mary Poppendieck",
      "number": "39-23-6423122"
    }
]
```

Output in the browser after GET request:



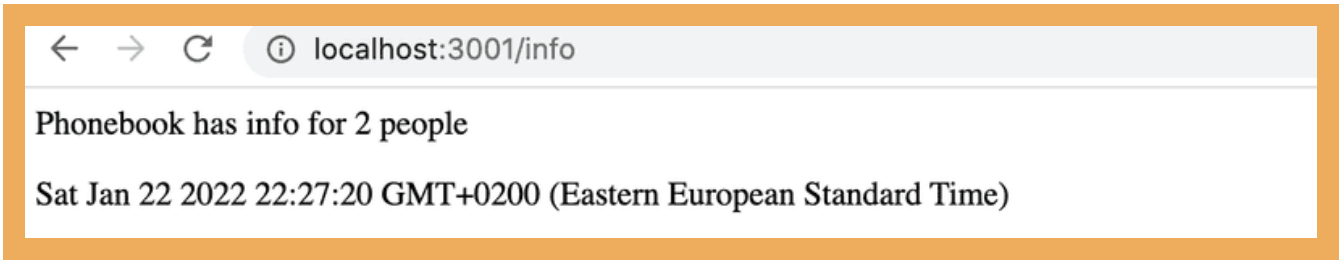
Notice that the forward slash in the route *api/persons* is not a special character, and is just like any other character in the string.

The application must be started with the command `npm start`.

The application must also offer an `npm run dev` command that will run the application and restart the server whenever changes are made and saved to a file in the source code.

3.2: Phonebook backend step 2

Implement a page at the address http://localhost:3001/info that looks roughly like this:



The page has to show the time that the request was received and how many entries are in the phonebook at the time of processing the request.

There can only be one response.send() statement in an Express app route. Once you send a response to the client using response.send(), the request-response cycle is complete and no further response can be sent.

To include a line space in the output, use `<br/>` tag, or wrap the statements in `<p>` tags.

3.3: Phonebook backend step 3

Implement the functionality for displaying the information for a single phonebook entry. The url for getting the data for a person with the id 5 should be http://localhost:3001/api/persons/5

If an entry for the given id is not found, the server has to respond with the appropriate status code.

3.4: Phonebook backend step 4

Implement functionality that makes it possible to delete a single phonebook entry by making an HTTP DELETE request to the unique URL of that phonebook entry.

Test that your functionality works with either Postman or the Visual Studio Code REST client.

3.5: Phonebook backend step 5

Expand the backend so that new phonebook entries can be added by making HTTP POST requests to the address http://localhost:3001/api/persons.

Generate a new id for the phonebook entry with the Math.random function. Use a big enough range for your random values so that the likelihood of creating duplicate ids is small.

3.6: Phonebook backend step 6

Implement error handling for creating new entries. The request is not allowed to succeed, if:

- The name or number is missing
- The name already exists in the phonebook

Respond to requests like these with the appropriate status code, and also send back information that explains the reason for the error, e.g.:

```
{ error: 'name must be unique' }
```

copy

About HTTP request types

The HTTP standard talks about two properties related to request types, **safety** and **idempotency**.

The HTTP GET request should be *safe*:

> *In particular, the convention has been established that the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered "safe".*

Safety means that the executing request must not cause any *side effects* on the server. By side effects, we mean that the state of the database must not change as a result of the request, and the response must only return data that already exists on the server.

Nothing can ever guarantee that a GET request is *safe*, this is just a recommendation that is defined in the HTTP standard. By adhering to RESTful principles in our API, GET requests are always used in a way that they are *safe*.

The HTTP standard also defines the request type HEAD, which ought to be safe. In practice, HEAD should work exactly like GET but it does not return anything but the status code and response headers. The response body will not be returned when you make a HEAD request.

All HTTP requests except POST should be *idempotent*:

> *Methods can also have the property of "idempotence" in that (aside from error or expiration issues) the side-effects of N > 0 identical requests is the same as for a single request. The methods GET, HEAD, PUT and DELETE share this property*

This means that if a request does not generate side effects, then the result should be the same regardless of how many times the request is sent.

If we make an HTTP PUT request to the URL */api/notes/10* and with the request we send the data `{ content: "no side effects!", important: true }`, the result is the same regardless of how many times the request is sent.

Like *safety* for the GET request, *idempotence* is also just a recommendation in the HTTP standard and not something that can be guaranteed simply based on the request type. However, when our API adheres to RESTful principles, then GET, HEAD, PUT, and DELETE requests are used in such a way that they are idempotent.

POST is the only HTTP request type that is neither *safe* nor *idempotent*. If we send 5 different HTTP POST requests to */api/notes* with a body of `{content: "many same", important: true}`, the resulting 5 notes on the server will all have the same content.

Middleware

The Express json-parser used earlier is a middleware.

Middleware are functions that can be used for handling `request` and `response` objects.

The json-parser we used earlier takes the raw data from the requests that are stored in the `request` object, parses it into a JavaScript object and assigns it to the `request` object as a new property *body*.

In practice, you can use several middlewares at the same time. When you have more than one, they're executed one by one in the order that they were listed in the application code.

Let's implement our own middleware that prints information about every request that is sent to the server.

Middleware is a function that receives three parameters:

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

copy

At the end of the function body, the `next` function that was passed as a parameter is called. The `next` function yields control to the next middleware.

Middleware is used like this:

```
app.use(requestLogger)
```

copy

Remember, middleware functions are called in the order that they're encountered by the JavaScript engine. Notice that `json-parser` is listed before `requestLogger` , because otherwise *request.body* will not be initialized when the logger is executed!

Middleware functions have to be used before routes when we want them to be executed by the route event handlers. Sometimes, we want to use middleware functions after routes. We do this when the middleware functions are only called if no route handler processes the HTTP request.

Let's add the following middleware after our routes. This middleware will be used for catching requests made to non-existent routes. For these requests, the middleware will return an error message in the JSON format.

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

app.use(unknownEndpoint)
```

You can find the code for our current application in its entirety in the *part3-2* branch of this GitHub repository.

Exercises 3.7.-3.8.

3.7: Phonebook backend step 7

Add the morgan middleware to your application for logging. Configure it to log messages to your console based on the *tiny* configuration.

The documentation for Morgan is not the best, and you may have to spend some time figuring out how to configure it correctly. However, most documentation in the world falls under the same category, so it's good to learn to decipher and interpret cryptic documentation in any case.

Morgan is installed just like all other libraries with the `npm install` command. Taking morgan into use happens the same way as configuring any other middleware by using the `app.use` command.

3.8*: Phonebook backend step 8

Configure morgan so that it also shows the data sent in HTTP POST requests:

```
Server running on port 3001
POST /api/persons 200 61 - 4.896 ms {"name":"Liisa Marttinen","number":"040-243563"}
```

Note that logging data even in the console can be dangerous since it can contain sensitive data and may violate local privacy law (e.g. GDPR in EU) or business-standard. In this exercise, you don't have to worry about privacy issues, but in practice, try not to log any sensitive data.

This exercise can be quite challenging, even though the solution does not require a lot of code.

This exercise can be completed in a few different ways. One of the possible solutions utilizes these two techniques:

- creating new tokens
- JSON.stringify

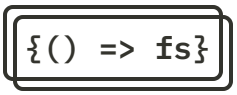
Propose changes to material

Part 2
Previous part

Part 3b
Next part

About course


HOUSTON



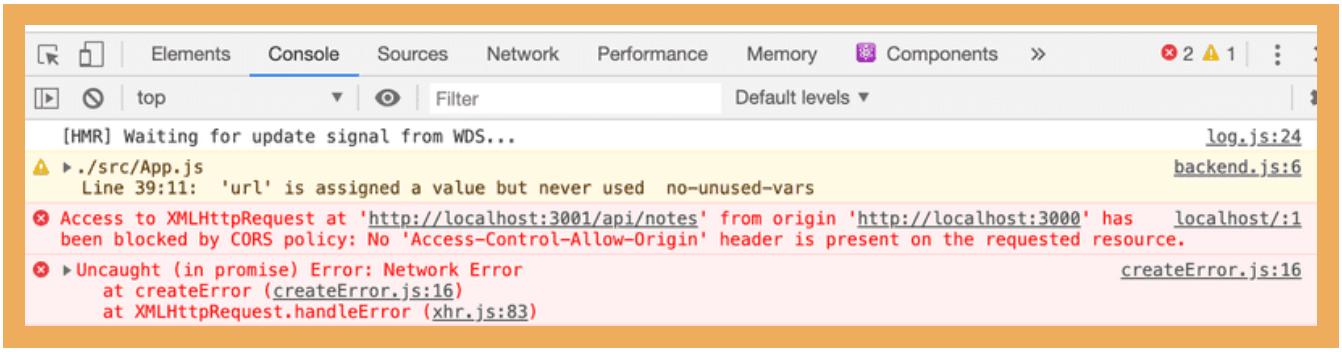
b Deploying app to internet

Next, let's connect the frontend we made in part 2 to our own backend.

In the previous part, the frontend could ask for the list of notes from the json-server we had as a backend, from the address http://localhost:3001/notes. Our backend has a slightly different URL structure now, as the notes can be found at http://localhost:3001/api/notes. Let's change the attribute **baseUrl** in the frontend notes app at *src/services/notes.js* like so:

```
import axios from 'axios'
const baseUrl = 'http://localhost:3001/api/notes'

const getAll = () => {
  const request = axios.get(baseUrl)
  return request.then(response => response.data)
}

// ...

export default { getAll, create, update }
```

Now frontend's GET request to http://localhost:3001/api/notes does not work for some reason:



What's going on here? We can access the backend from a browser and from postman without any problems.

Same origin policy and CORS

The issue lies with a thing called `same origin policy`. A URL's origin is defined by the combination of protocol (AKA scheme), hostname, and port.

```
http://example.com:80/index.html

protocol: http
host: example.com
port: 80
```

When you visit a website (e.g. http://catwebsites.com), the browser issues a request to the server on which the website (catwebsites.com) is hosted. The response sent by the server is an HTML file that may contain one or more references to external assets/resources hosted either on the same server that *catwebsites.com* is hosted on or a different website. When the browser sees reference(s) to a URL in the source HTML, it issues a request. If the request is issued using the URL that the source HTML was fetched from, then the browser processes the response without any issues. However, if the resource is fetched using a URL that doesn't share the same origin(scheme, host, port) as the source HTML, the browser will have to check the `Access-Control-Allow-origin` response header. If it contains `*` on the URL of the source HTML, the browser will process the response, otherwise the browser will refuse to process it and throws an error.

The **same-origin policy** is a security mechanism implemented by browsers in order to prevent session hijacking among other security vulnerabilities.

In order to enable legitimate cross-origin requests (requests to URLs that don't share the same origin) W3C came up with a mechanism called **CORS**(Cross-Origin Resource Sharing). According to Wikipedia:

> *Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources (e.g. fonts) on a web page to be requested from another domain outside the domain from which the first resource was served. A web page may freely embed cross-origin images, stylesheets, scripts, iframes, and videos. Certain "cross-domain" requests, notably Ajax requests, are forbidden by default by the same-origin security policy.*

The problem is that, by default, the JavaScript code of an application that runs in a browser can only communicate with a server in the same origin. Because our server is in localhost port 3001, while our frontend is in localhost port 5173, they do not have the same origin.

Keep in mind, that same-origin policy and CORS are not specific to React or Node. They are universal principles regarding the safe operation of web applications.

We can allow requests from other *origins* by using Node's cors middleware.

In your backend repository, install *cors* with the command

```
npm install cors
```

copy

take the middleware to use and allow for requests from all origins:

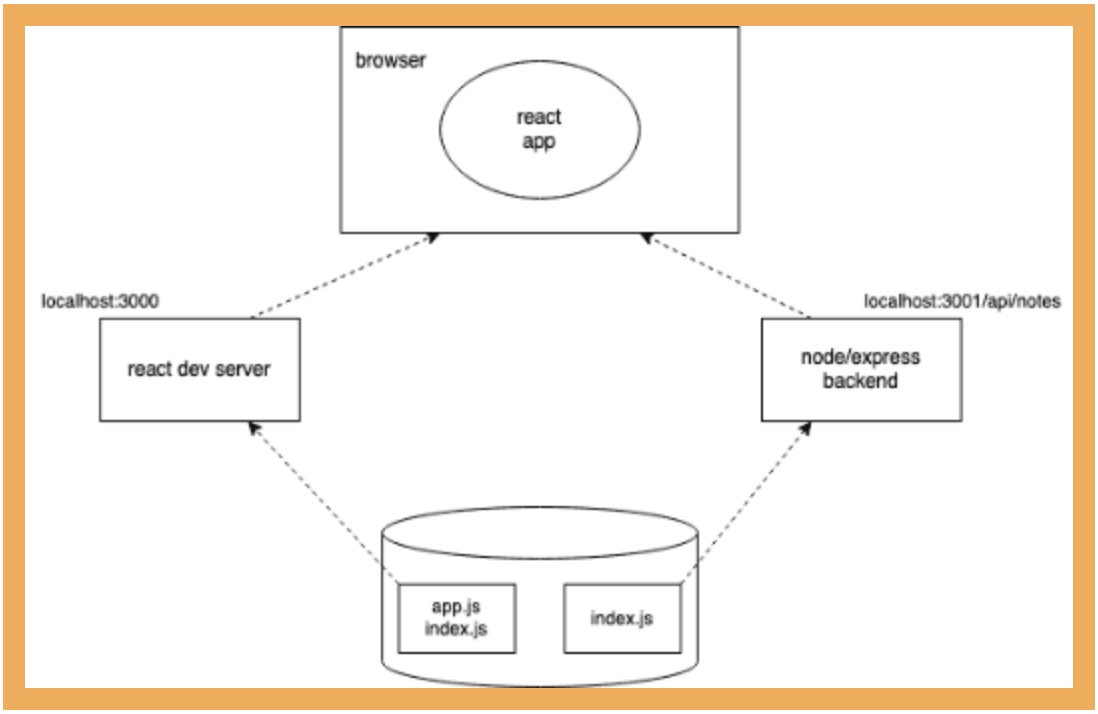
```
const cors = require('cors')

app.use(cors())
```

copy

Now most of the features in the frontend work! The functionality for changing the importance of notes has not yet been implemented on the backend so naturally that does not yet work in the frontend. We shall fix that later.

You can read more about CORS from Mozilla's page.

The setup of our app looks now as follows:



The react app running in the browser now fetches the data from node/express-server that runs in localhost:3001.

Application to the Internet

Now that the whole stack is ready, let's move our application to Internet.

There is an ever-growing number of services that can be used to host an app on the internet. The developer-friendly services like PaaS (i.e. Platform as a Service) take care of installing the execution environment (eg. Node.js) and could also provide various services such as databases.

For a decade, Heroku was dominating the PaaS scene. Unfortunately the free tier Heroku ended at 27th November 2022. This is very unfortunate for many developers, especially students. Heroku is still very much a viable option if you are willing to spend some money. They also have a student program that provides some free credits.

We are now introducing two services Fly.io and Render that both have a (limited) free plan. Fly.io is our "official" hosting service since it can be for sure used also on parts 11 and 13 of the course. Render will be fine at least for the other parts of this course.

Note that despite using the free tier only, Fly.io *might* require one to enter their credit card details. At the moment Render can be used without a credit card.

Render might be a bit easier to use since it does not require any software to be installed on your machine.

There are also some other free hosting options that work well for this course, at least for all parts other than part 11 (CI/CD) which might have one tricky exercise for other platforms.

Some course participants have also used the following services:

- Cyclic
- Replit
- CodeSandBox

If you know some other good and easy-to-use services for hosting NodeJS, please let us know!

For both Fly.io and Render, we need to change the definition of the port our application uses at the bottom of the *index.js* file in the backend like so:

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

copy

Now we are using the port defined in the environment variable `PORT` or port 3001 if the environment variable `PORT` is undefined. Fly.io and Render configure the application port based on that environment variable.

Fly.io

Note that you may need to give your credit card number to Fly.io even if you are using only the free tier! There has been actually conflicting reports about this, it is known for a fact that some of the students in this course are using Fly.io without entering their credit card info. At the moment Render can be used without a credit card.

By default, everyone gets two free virtual machines that can be used for running two apps at the same time.

If you decide to use Fly.io begin by installing their flyctl executable following this guide. After that, you should create a Fly.io account.

Start by authenticating via the command line with the command

```
fly auth login
```

copy

Note if the command `fly` does not work on your machine, you can try the longer version `flyctl`. Eg. on MacOS, both forms of the command work.

If you do not get the flyctl to work in your machine, you could try Render (see next section), it does not require anything to be installed in your machine.

Initializing an app happens by running the following command in the root directory of the app

```
fly launch
```

copy

Give the app a name or let Fly.io auto-generate one. Pick a region where the app will be run. Do not create a Postgres database for the app and do not create an Upstash Redis database, since these are not needed.

The last question is "Would you like to deploy now?". We should answer "no" since we are not quite ready yet.

Fly.io creates a file *fly.toml* in the root of your app where we can configure it. To get the app up and running we *might* need to do a small addition to the configuration:

```
[build]

[env]
  PORT = "3000" # add this
```

copy

```
[http_service]
  internal_port = 3000 # ensure that this is same as PORT
  force_https = true
  auto_stop_machines = true
  auto_start_machines = true
  min_machines_running = 0
  processes = ["app"]
```

We have now defined in the part [env] that environment variable PORT will get the correct port (defined in part [http_service]) where the app should create the server.

We are now ready to deploy the app to the Fly.io servers. That is done with the following command:

```
fly deploy
```

copy

If all goes well, the app should now be up and running. You can open it in the browser with the command

```
fly apps open
```

copy

A particularly important command is `fly logs`. This command can be used to view server logs. It is best to keep logs always visible!

Note: Fly may create 2 machines for your app, if it does then the state of the data in your app will be inconsistent between requests, i.e. you would have two machines each with its own notes variable, you could POST to one machine then your next GET could go to another machine. You can check the number of machines by using the command "$ fly scale show", if the COUNT is greater than 1 then you can enforce it to be 1 with the command "$ fly scale count 1". The machine count can also be checked on the dashboard.

Note: In some cases (the cause is so far unknown) running Fly.io commands especially on Windows WSL (Windows Subsystem for Linux) has caused problems. If the following command just hangs

```
flyctl ping -o personal
```

copy

your computer can not for some reason connect to Fly.io. If this happens to you, this describes one possible way to proceed.

If the output of the below command looks like this:

```
$ flyctl ping -o personal
35 bytes from fdaa:0:8a3d::3 (gateway), seq=0 time=65.1ms
```

copy

```
35 bytes from fdaa:0:8a3d::3 (gateway), seq=1 time=28.5ms
35 bytes from fdaa:0:8a3d::3 (gateway), seq=2 time=29.3ms
...
```

then there are no connection problems!

Whenever you make changes to the application, you can take the new version to production with a command

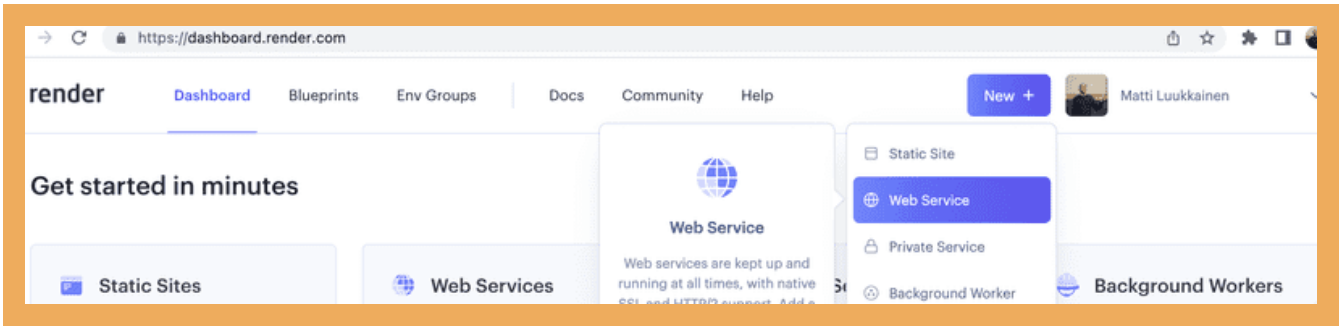
```
fly deploy
```

copy

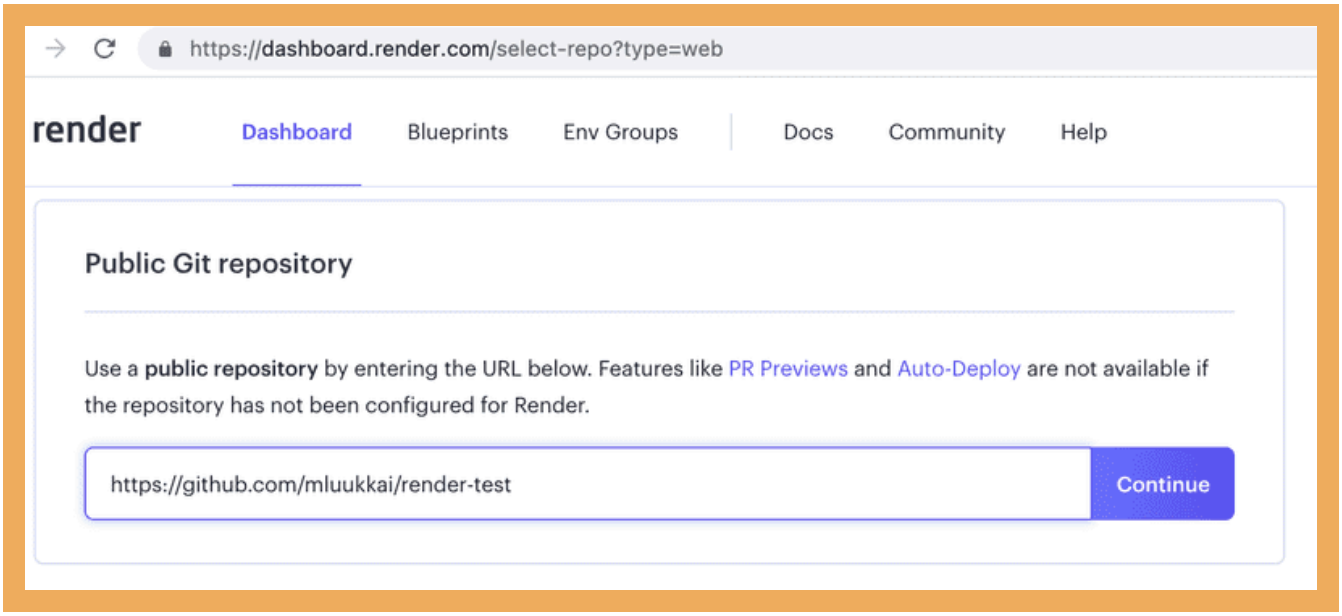
Render

The following assumes that the sign in has been made with a GitHub account.

After signing in, let us create a new "web service":

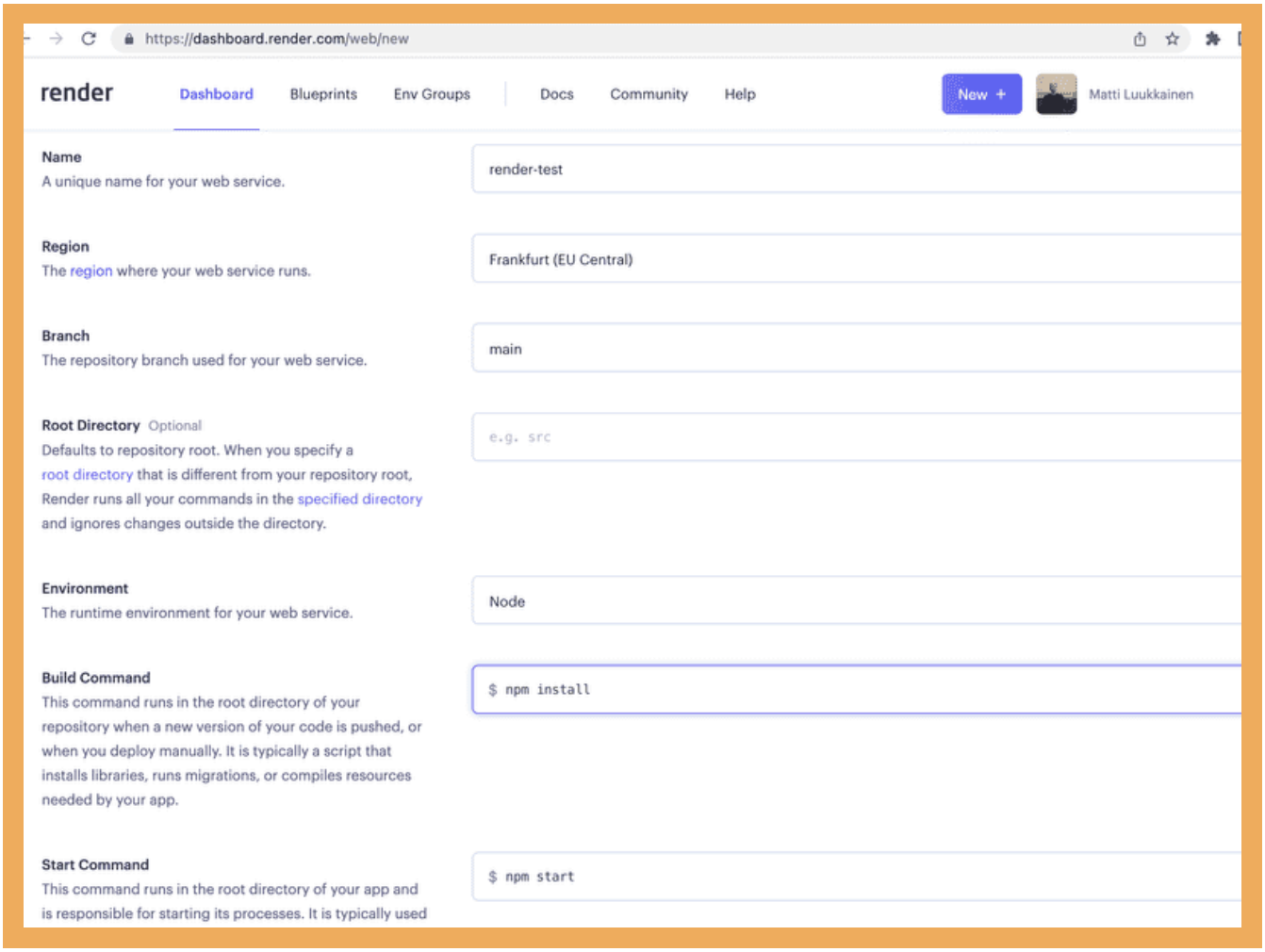


The app repository is then connected to Render:

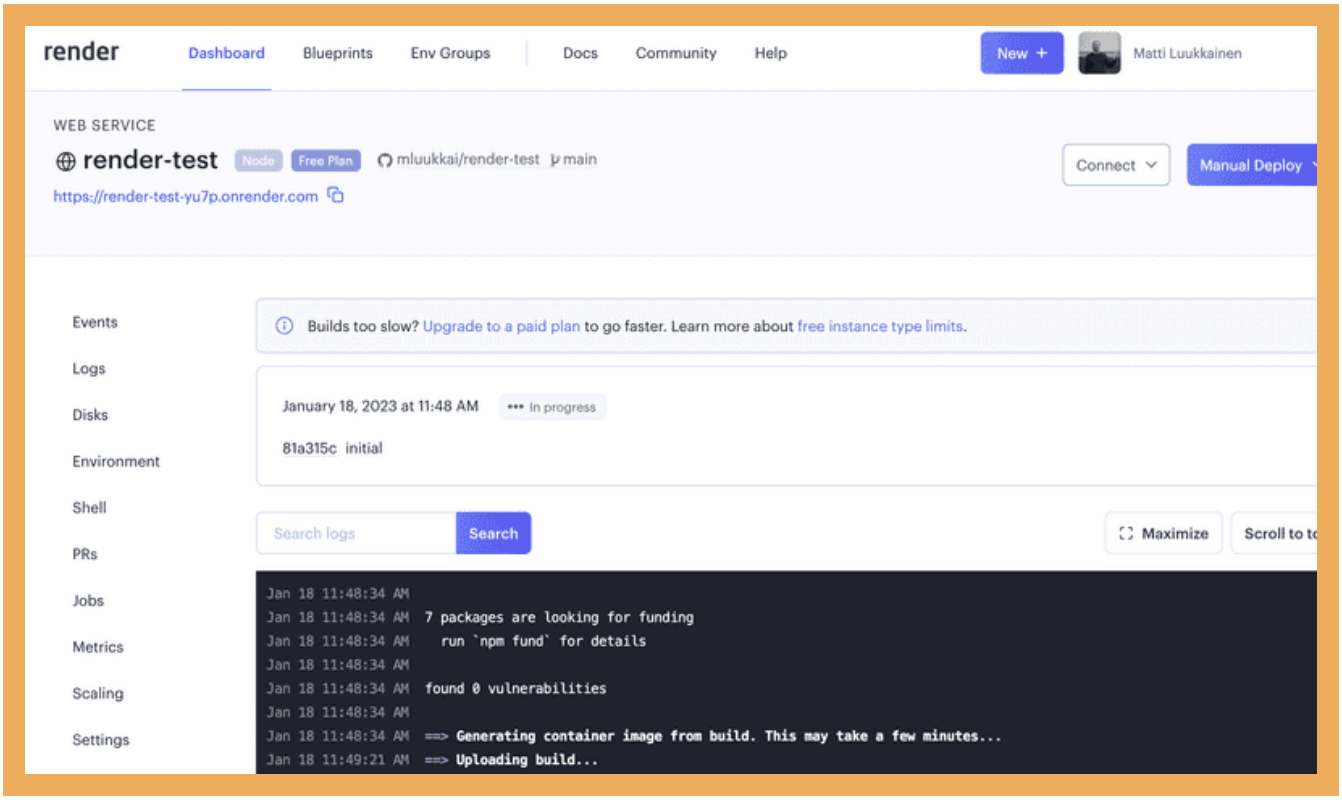


The connection seems to require that the app repository is public.

Next we will define the basic configurations. If the app is *not* at the root of the repository the *Root directory* needs to be given a proper value:

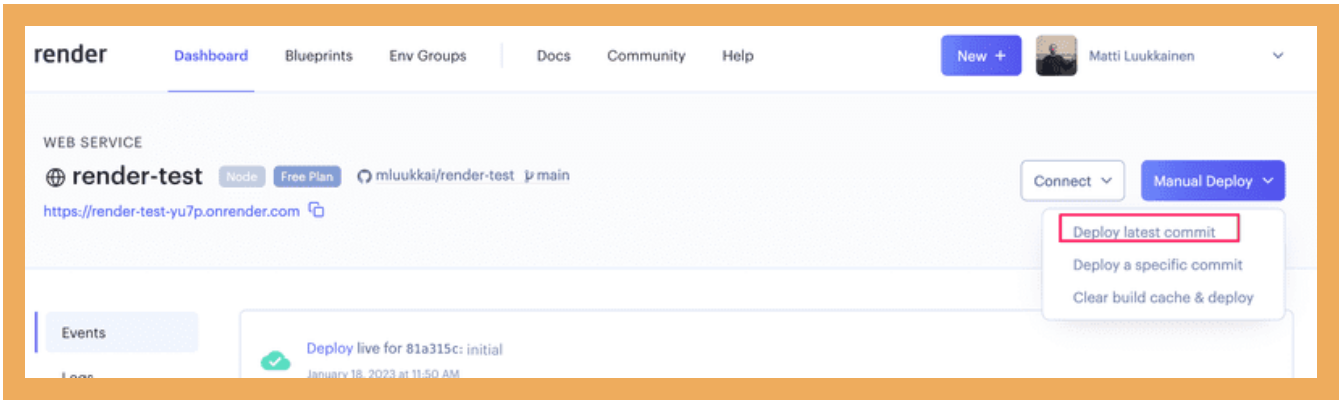


After this, the app starts up in the Render. The dashboard tells us the app state and the url where the app is running:

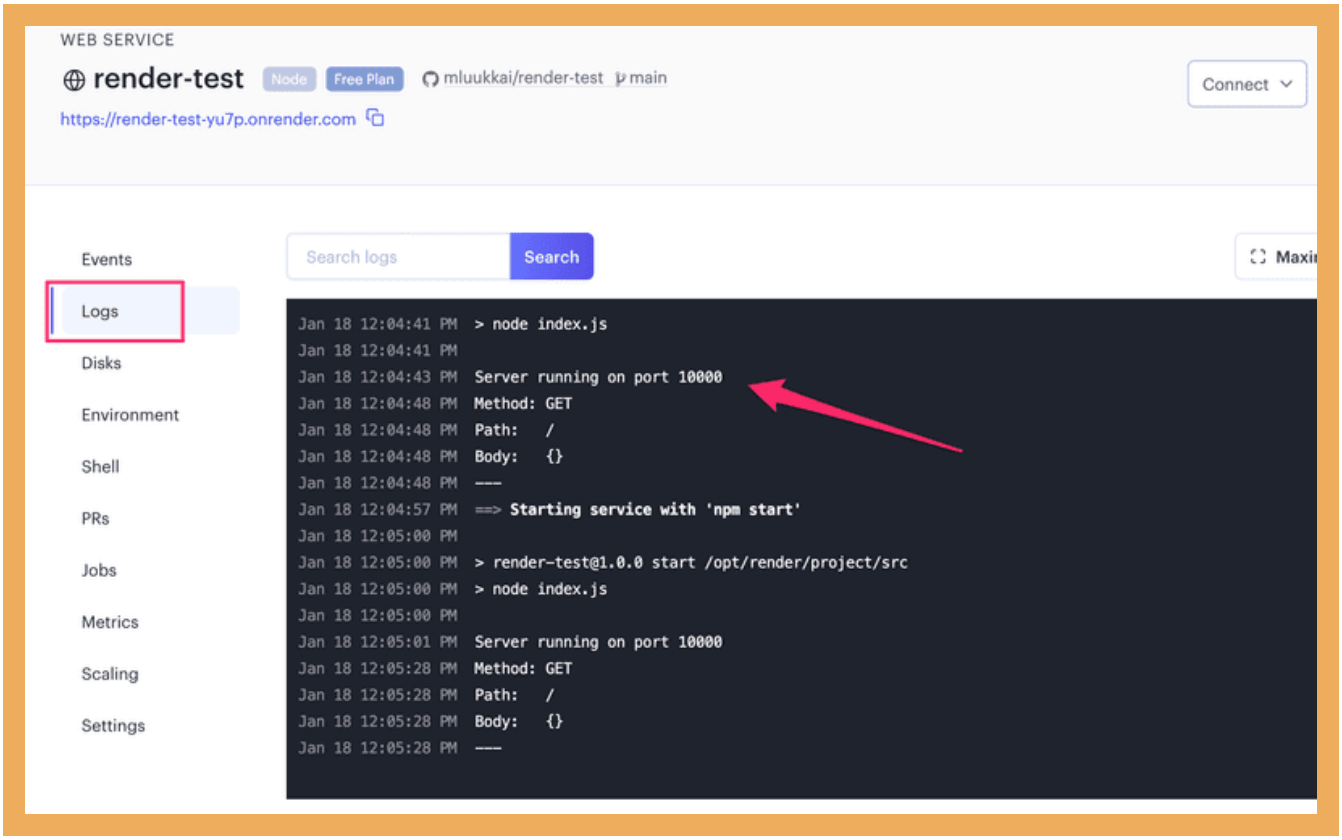


According to the documentation every commit to GitHub should redeploy the app. For some reason this is not always working.

Fortunately, it is also possible to manually redeploy the app:



Also, the app logs can be seen in the dashboard:



We notice now from the logs that the app has been started in the port 10000. The app code gets the right port through the environment variable PORT so it is essential that the file *index.js* has been updated in the backend as follows:

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

Frontend production build

So far we have been running React code in *development mode*. In development mode the application is configured to give clear error messages, immediately render code changes to the browser, and so on.

When the application is deployed, we must create a production build or a version of the application that is optimized for production.

A production build for applications created with Vite can be created with the command npm run build.

Let's run this command from the *root of the notes frontend project* that we developed in Part 2.

This creates a directory called *dist* which contains the only HTML file of our application (*index.html*) and the directory *assets*. Minified version of our application's JavaScript code will be generated in the *dist* directory. Even though the application code is in multiple files, all of the JavaScript will be minified into

one file. All of the code from all of the application's dependencies will also be minified into this single file.

The minified code is not very readable. The beginning of the code looks like this:

```
!function(e){function r(r){for(var n,f,i=r[0],l=r[1],a=r[2],c=0,s=
[];c<i.length;c++)f=i[c],o[f]&&s.push(o[f][0]),o[f]=0;for(n in
l)Object.prototype.hasOwnProperty.call(l,n)&&(e[n]=l[n]);for(p&&p(r);s.length;)s.shift()
();return u.push.apply(u,a||[]),t()}function t(){for(var e,r=0;r<u.length;r++){for(var
t=u[r],n=!0,i=1;i<t.length;i++){var l=t[i];0!==o[l]&&(n=!1)}n&&(u.splice(r-
-,1),e=f(f.s=t[0]))}return e}var n={},o={2:0},u=[];function f(r){if(n[r])return
n[r].exports;var t=n[r]={i:r,l:!1,exports:{}};return
e[r].call(t.exports,t,t.exports,f),t.l=!0,t.exports}f.m=e,f.c=n,f.d=function(e,r,t)
{f.o(e,r)||Object.defineProperty(e,r,{enumerable:!0,get:t})},f.r=function(e)
{"undefined"!==typeof
Symbol&&Symbol.toStringTag&&Object.defineProperty(e,Symbol.toStringTag,{value:"Module"})
```

Serving static files from the backend

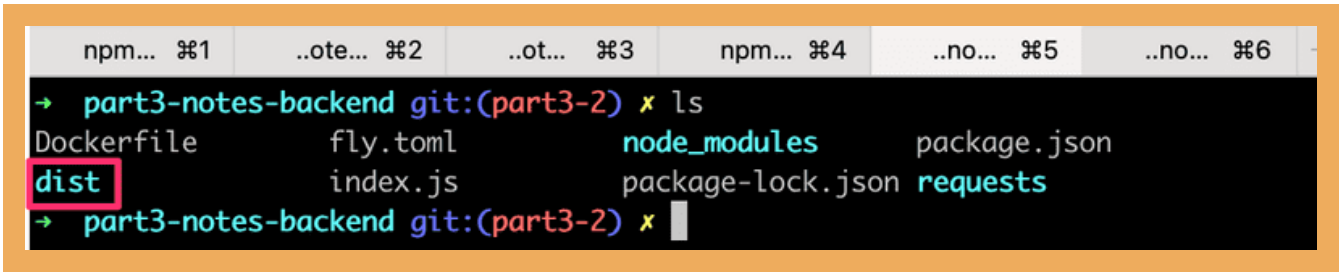
One option for deploying the frontend is to copy the production build (the *dist* directory) to the root of the backend repository and configure the backend to show the frontend's *main page* (the file *dist/index.html*) as its main page.

We begin by copying the production build of the frontend to the root of the backend. With a Mac or Linux computer, the copying can be done from the frontend directory with the command

```
cp -r dist ../backend
```

If you are using a Windows computer, you may use either `copy` or `xcopy` command instead. Otherwise, simply copy and paste.

The backend directory should now look as follows:



To make Express show *static content*, the page *index.html* and the JavaScript, etc., it fetches, we need a built-in middleware from Express called `static`.

When we add the following amidst the declarations of middlewares

```
app.use(express.static('dist'))
```

copy

whenever Express gets an HTTP GET request it will first check if the *dist* directory contains a file corresponding to the request's address. If a correct file is found, Express will return it.

Now HTTP GET requests to the address *www.serversaddress.com/index.html* or *www.serversaddress.com* will show the React frontend. GET requests to the address *www.serversaddress.com/api/notes* will be handled by the backend code.

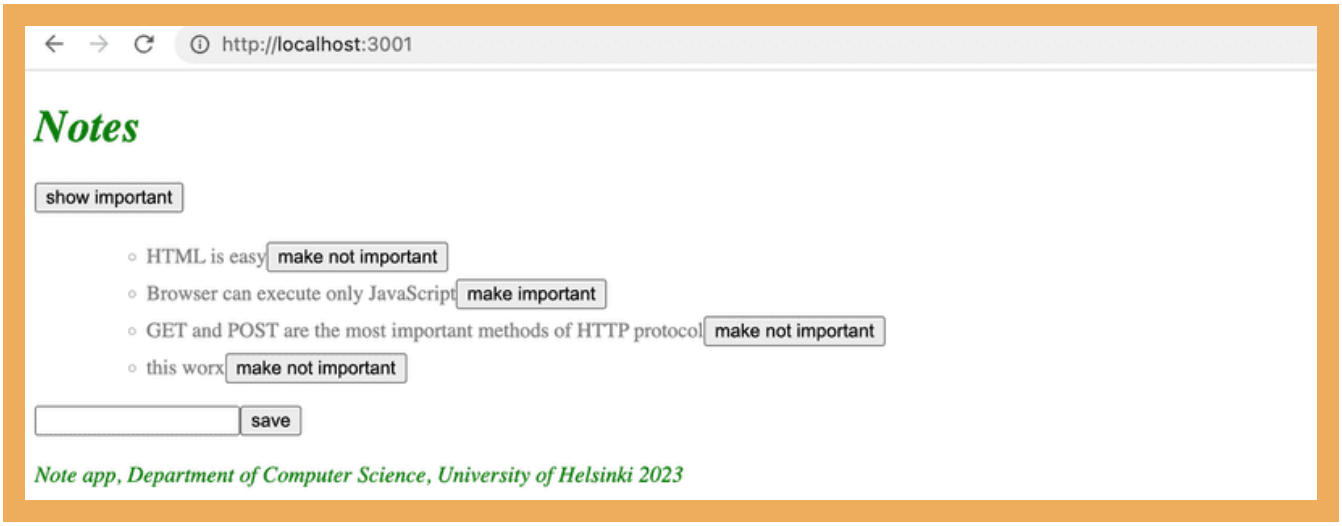
Because of our situation, both the frontend and the backend are at the same address, we can declare `baseUrl` as a relative URL. This means we can leave out the part declaring the server.

```
import axios from 'axios'
const baseUrl = '/api/notes'

const getAll = () => {
  const request = axios.get(baseUrl)
  return request.then(response => response.data)
}

// ...
```

copy

After the change, we have to create a new production build of the frontend and copy it to the root of the backend repository.

The application can now be used from the *backend* address http://localhost:3001:



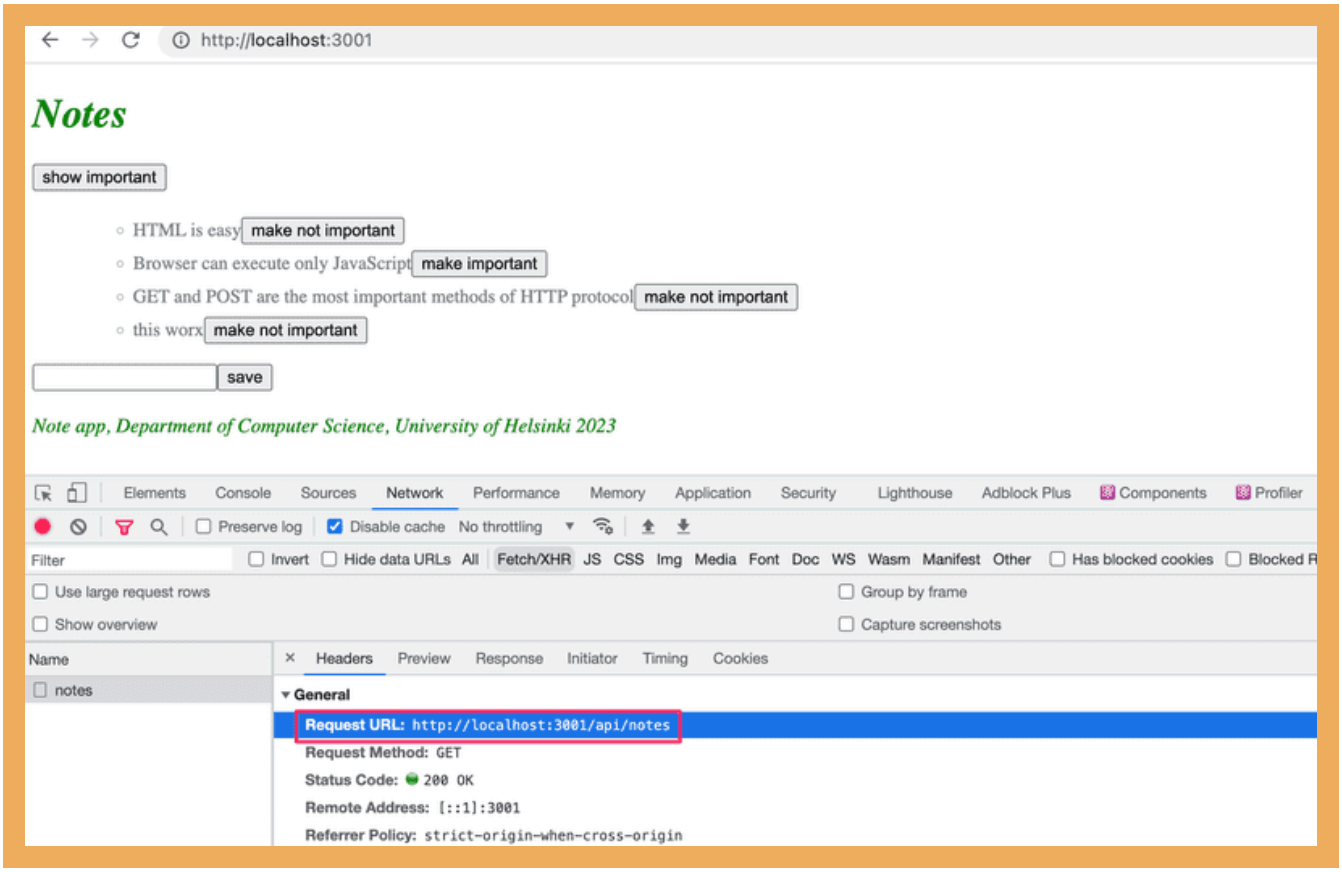
Our application now works exactly like the single-page app example application we studied in part 0.

When we use a browser to go to the address http://localhost:3001, the server returns the *index.html* file from the *dist* directory. The contents of the file are as follows:

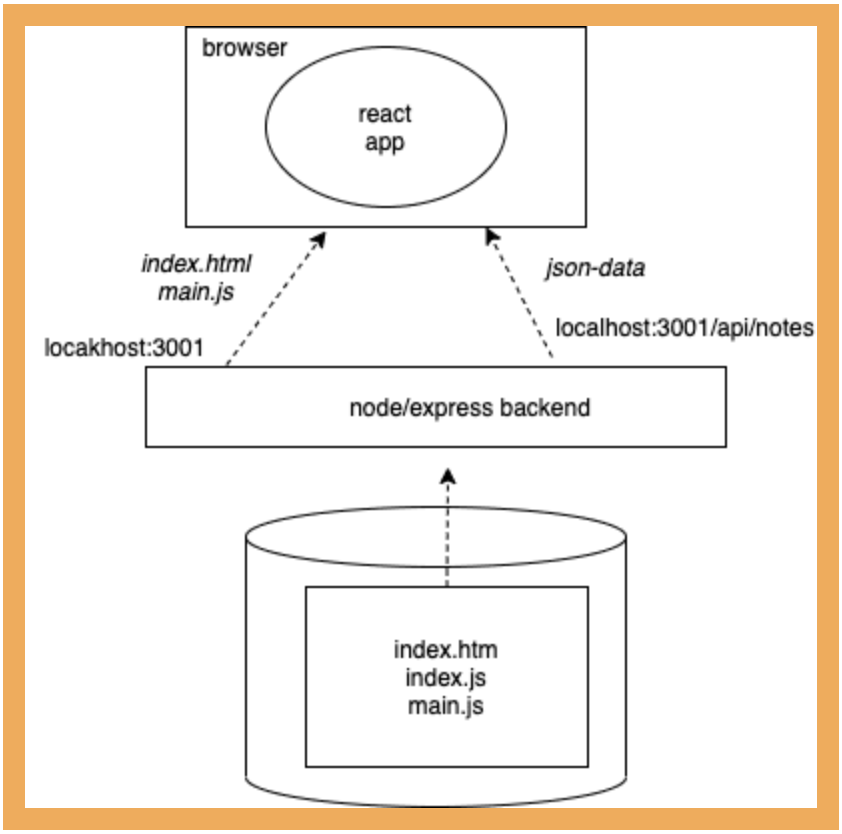
```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
    <script type="module" crossorigin src="/assets/index-5f6faa37.js"></script>
    <link rel="stylesheet" href="/assets/index-198af077.css">
  </head>
  <body>
    <div id="root"></div>

  </body>
</html>
```

The file contains instructions to fetch a CSS stylesheet defining the styles of the application, and one *script* tag that instructs the browser to fetch the JavaScript code of the application - the actual React application.

The React code fetches notes from the server address http://localhost:3001/api/notes and renders them to the screen. The communication between the server and the browser can be seen in the *Network* tab of the developer console:



The setup that is ready for a product deployment looks as follows:



Unlike when running the app in a development environment, everything is now in the same node/express-backend that runs in localhost:3001. When the browser goes to the page, the file *index.html* is rendered. That causes the browser to fetch the production version of the React app. Once it starts to run, it fetches the json-data from the address localhost:3001/api/notes.

The whole app to the internet

After ensuring that the production version of the application works locally, commit the production build of the frontend to the backend repository, and push the code to GitHub again.

NB If you use Render, make sure the directory *dist* is not ignored by git on the backend.

If you are using Render a push to GitHub *might* be enough. If the automatic deployment does not work, select the "manual deploy" from the Render dashboard.

In the case of Fly.io the new deployment is done with the command

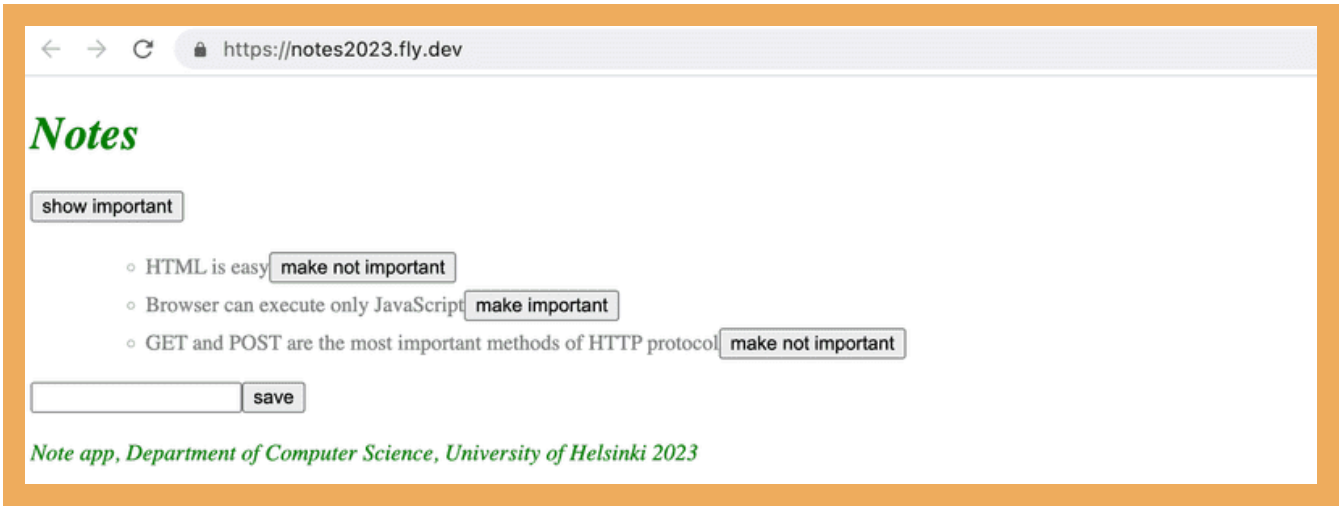
```
fly deploy
```

copy

The application works perfectly, except we haven't added the functionality for changing the importance of a note to the backend yet.

NOTE: When using Fly.io, be aware that the `.dockerignore` file in your project directory lists files not uploaded during deployment. The dist directory is included by default. To deploy this directory, remove

its reference from the .dockerignore file, ensuring your app is get properly deployed.

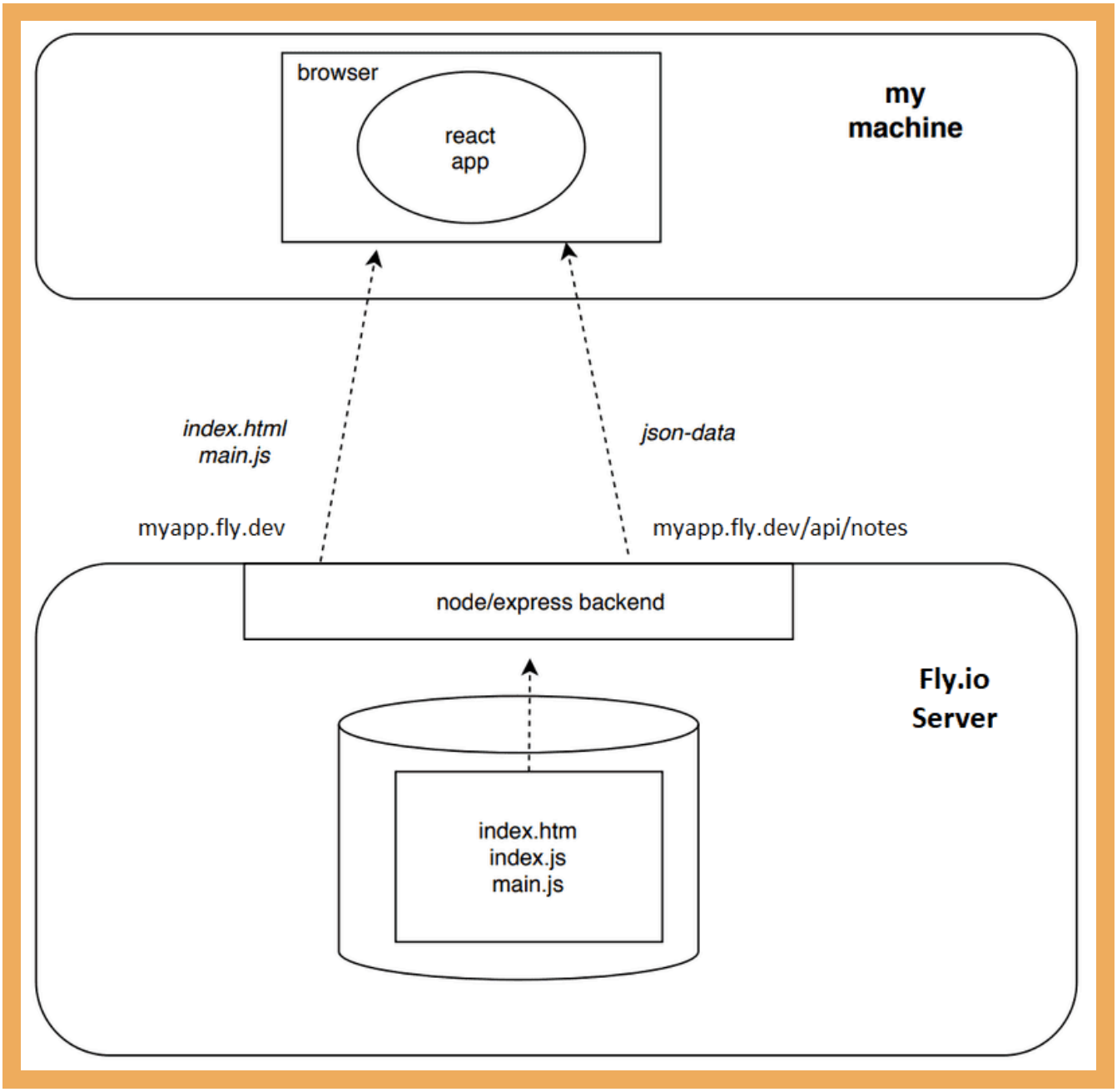


NOTE: *changing the importance DOES NOT work yet since the backend has no implementation for it yet.*

Our application saves the notes to a variable. If the application crashes or is restarted, all of the data will disappear.

The application needs a database. Before we introduce one, let's go through a few things.

The setup now looks like as follows:



The node/express-backend now resides in the Fly.io/Render server. When the root address is accessed, the browser loads and executes the React app that fetches the json-data from the Fly.io/Render server.

Streamlining deploying of the frontend

To create a new production build of the frontend without extra manual work, let's add some npm-scripts to the *package.json* of the backend repository.

Fly.io script

The scripts look like this:

```
{
  "scripts": {
    // ...
    "build:ui": "rm -rf dist && cd ../notes-frontend/ && npm run build && cp -r dist
../notes-backend",
    "deploy": "fly deploy",
    "deploy:full": "npm run build:ui && npm run deploy",
    "logs:prod": "fly logs"
  }
}
```

Note for Windows users

Note that the standard shell commands in `build:ui` do not natively work in Windows. Powershell in Windows works differently, in which case the script could be written as

```
"build:ui": "@powershell Remove-Item -Recurse -Force dist && cd ../frontend && npm run
build && @powershell Copy-Item dist -Recurse ../backend",
```

If the script does not work on Windows, confirm that you are using Powershell and not Command Prompt. If you have installed Git Bash or another Linux-like terminal, you may be able to run Linux-like commands on Windows as well.

The script `npm run build:ui` builds the frontend and copies the production version under the backend repository. The script `npm run deploy` releases the current backend to Fly.io.

`npm run deploy:full` combines these two scripts, i.e., `npm run build:ui` and `npm run deploy`.

There is also a script `npm run logs:prod` to show the Fly.io logs.

Note that the directory paths in the script *build:ui* depend on the location of repositories in the file system.

Render

Note: When you attempt to deploy your backend to Render, make sure you have a separate repository for the backend and deploy that github repo through Render, attempting to deploy through your Fullstackopen repository will often throw "ERR path ....package.json".

In case of Render, the scripts look like the following

```
{
  "scripts": {
    //...
    "build:ui": "rm -rf dist && cd ../frontend && npm run build && cp -r dist
```

```
../backend",
    "deploy:full": "npm run build:ui && git add . && git commit -m uibuild && git push"
  }
}
```

The script `npm run build:ui` builds the frontend and copies the production version under the backend repository. `npm run deploy:full` contains also the necessary *git* commands to update the backend repository.

Note that the directory paths in the script *build:ui* depend on the location of repositories in the file system.

> **NB** On Windows, npm scripts are executed in cmd.exe as the default shell which does not support bash commands. For the above bash commands to work, you can change the default shell to Bash (in the default Git for Windows installation) as follows:

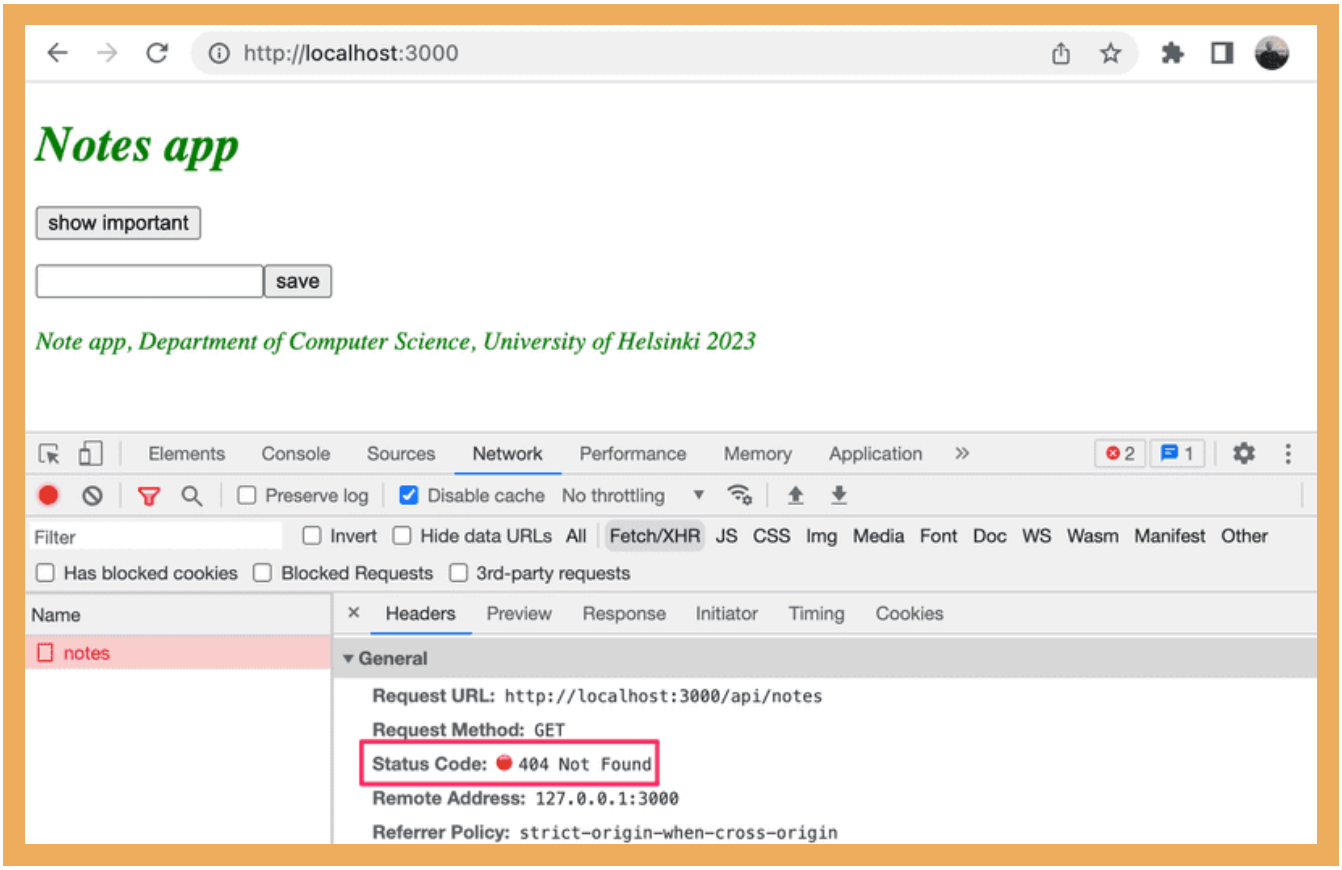
```
npm config set script-shell "C:\\Program Files\\git\\bin\\bash.exe"
```

copy

Another option is the use of shx.

Proxy

Changes on the frontend have caused it to no longer work in development mode (when started with command `npm run dev`), as the connection to the backend does not work.



This is due to changing the backend address to a relative URL:

```
const baseUrl = '/api/notes'
```

Because in development mode the frontend is at the address *localhost:5173*, the requests to the backend go to the wrong address *localhost:5173/api/notes*. The backend is at *localhost:3001*.

If the project was created with Vite, this problem is easy to solve. It is enough to add the following declaration to the *vite.config.js* file of the frontend repository.

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
```

```
  server: {
    proxy: {
      '/api': {
        target: 'http://localhost:3001',
        changeOrigin: true,
      },
    }
  },
})
```

After a restart, the React development environment will work as a proxy. If the React code does an HTTP request to a server address at *http://localhost:5173* not managed by the React application itself (i.e. when requests are not about fetching the CSS or JavaScript of the application), the request will be redirected to the server at *http://localhost:3001*.

Note that with the vite-configuration shown above, only requests that are made to paths starting with */api*-are redirected to the server.

Now the frontend is also fine, working with the server both in development and production mode.

A negative aspect of our approach is how complicated it is to deploy the frontend. Deploying a new version requires generating a new production build of the frontend and copying it to the backend repository. This makes creating an automated deployment pipeline more difficult. Deployment pipeline means an automated and controlled way to move the code from the computer of the developer through different tests and quality checks to the production environment. Building a deployment pipeline is the topic of part 11 of this course. There are multiple ways to achieve this, for example, placing both backend and frontend code in the same repository but we will not go into those now.

In some situations, it may be sensible to deploy the frontend code as its own application.

The current backend code can be found on Github, in the branch *part3-3*. The changes in frontend code are in *part3-1* branch of the frontend repository.

Exercises 3.9.-3.11

The following exercises don't require many lines of code. They can however be challenging, because you must understand exactly what is happening and where, and the configurations must be just right.

3.9 Phonebook backend step 9

Make the backend work with the phonebook frontend from the exercises of the previous part. Do not implement the functionality for making changes to the phone numbers yet, that will be implemented in exercise 3.17.

You will probably have to do some small changes to the frontend, at least to the URLs for the backend. Remember to keep the developer console open in your browser. If some HTTP requests fail, you should check from the *Network*-tab what is going on. Keep an eye on the backend's console as well. If you did

not do the previous exercise, it is worth it to print the request data or *request.body* to the console in the event handler responsible for POST requests.

3.10 Phonebook backend step 10

Deploy the backend to the internet, for example to Fly.io or Render.

Test the deployed backend with a browser and Postman or VS Code REST client to ensure it works.

PRO TIP: When you deploy your application to Internet, it is worth it to at least in the beginning keep an eye on the logs of the application **AT ALL TIMES**.

Create a README.md at the root of your repository, and add a link to your online application to it.

NOTE: as it was said, 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.

3.11 Full Stack Phonebook

Generate a production build of your frontend, and add it to the Internet application using the method introduced in this part.

NB If you use Render, make sure the directory *dist* is not ignored by git on the backend.

Also, make sure that the frontend still works locally (in development mode when started with command `npm run dev`).

If you have problems getting the app working make sure that your directory structure matches the example app.

Propose changes to material

Part 3a
Previous part

Part 3c
Next part

About course

Course contents

FAQ

Partners

Challenge


HOUSTON

{() => fs}

Fullstack Part 3 Saving data to MongoDB

c Saving data to MongoDB

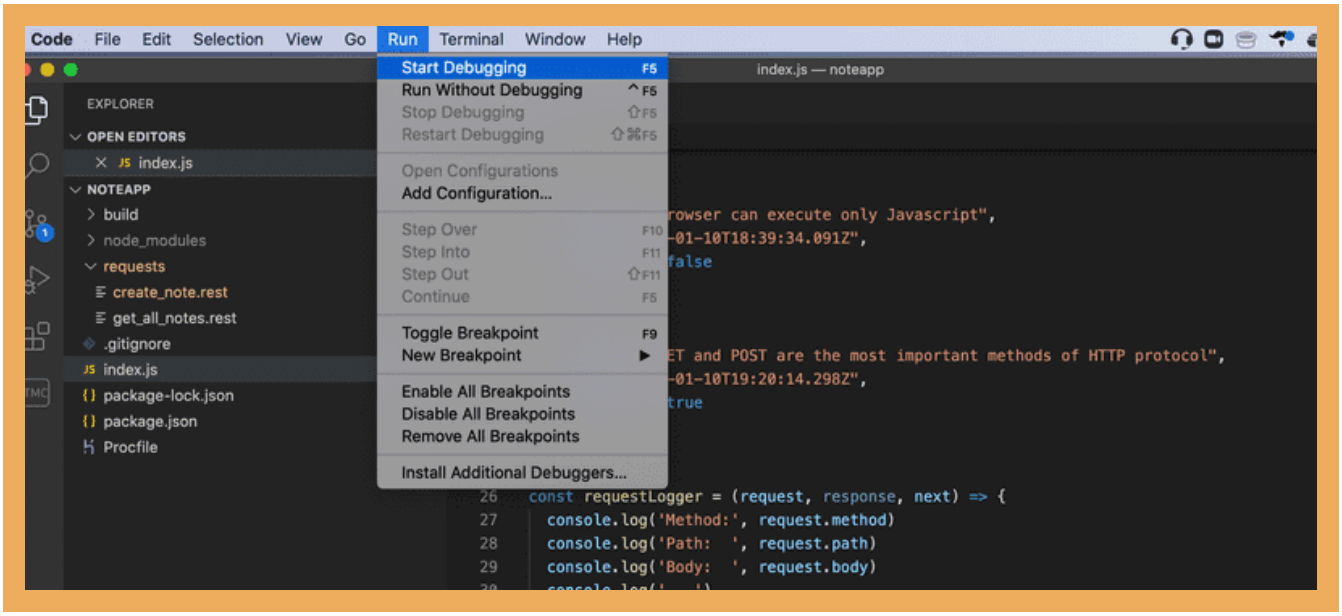
Before we move into the main topic of persisting data in a database, we will take a look at a few different ways of debugging Node applications.

Debugging Node applications

Debugging Node applications is slightly more difficult than debugging JavaScript running in your browser. Printing to the console is a tried and true method, and it's always worth doing. Some people think that more sophisticated methods should be used instead, but I disagree. Even the world's elite open-source developers use this method.

Visual Studio Code

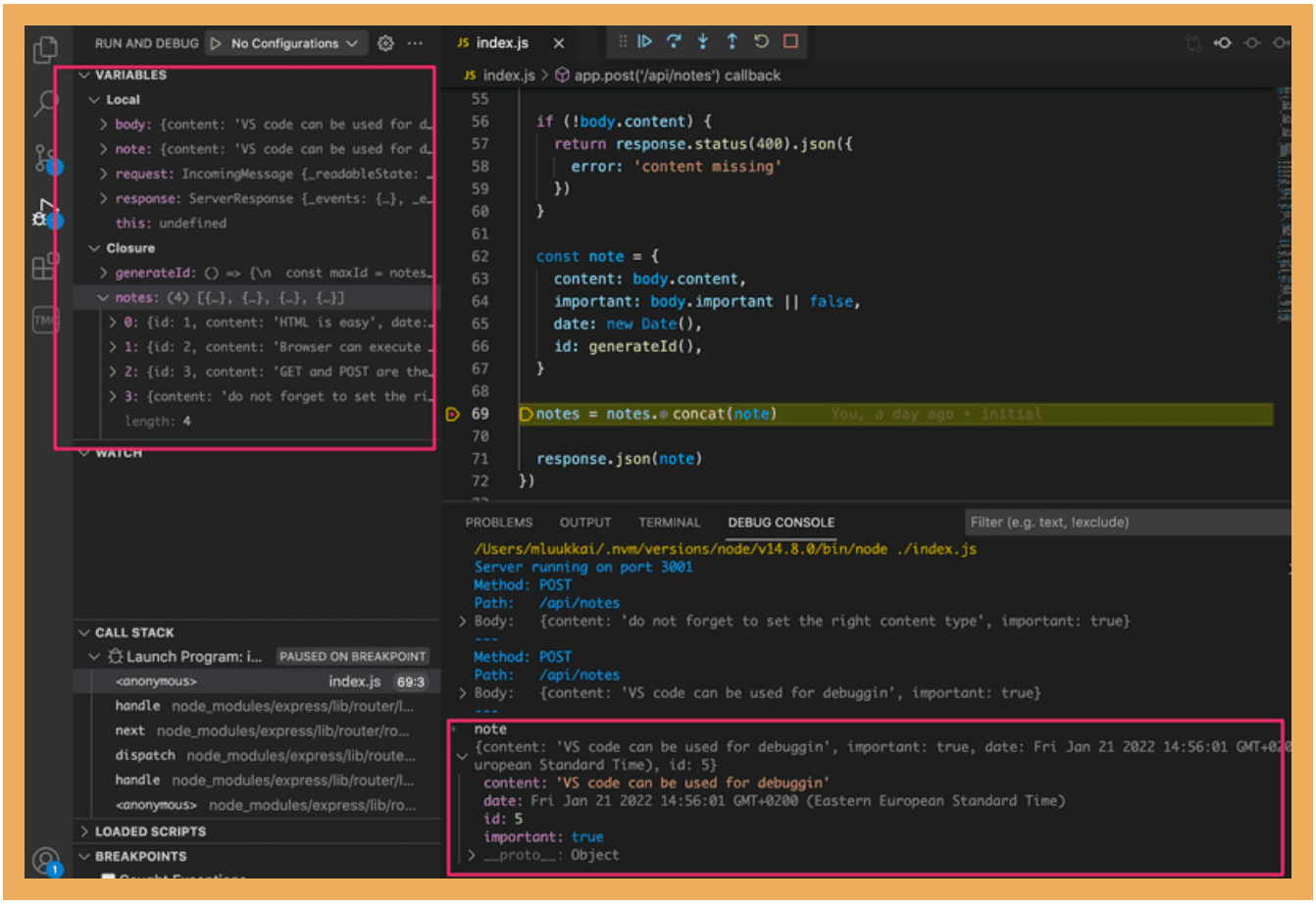
The Visual Studio Code debugger can be useful in some situations. You can launch the application in debugging mode like this (in this and the next few images, the notes have a field `date` which has been removed from the current version of the application):



Note that the application shouldn't be running in another console, otherwise the port will already be in use.

NB A newer version of Visual Studio Code may have `Run` instead of `Debug`. Furthermore, you may have to configure your `launch.json` file to start debugging. This can be done by choosing `Add Configuration...` on the drop-down menu, which is located next to the green play button and above `VARIABLES` menu, and select `Run "npm start" in a debug terminal`. For more detailed setup instructions, visit Visual Studio Code's Debugging documentation.

Below you can see a screenshot where the code execution has been paused in the middle of saving a new note:



The execution stopped at the *breakpoint* in line 69. In the console, you can see the value of the *note* variable. In the top left window, you can see other things related to the state of the application.

The arrows at the top can be used for controlling the flow of the debugger.

For some reason, I don't use the Visual Studio Code debugger a whole lot.

Chrome dev tools

Debugging is also possible with the Chrome developer console by starting your application with the command:

```
node --inspect index.js
```

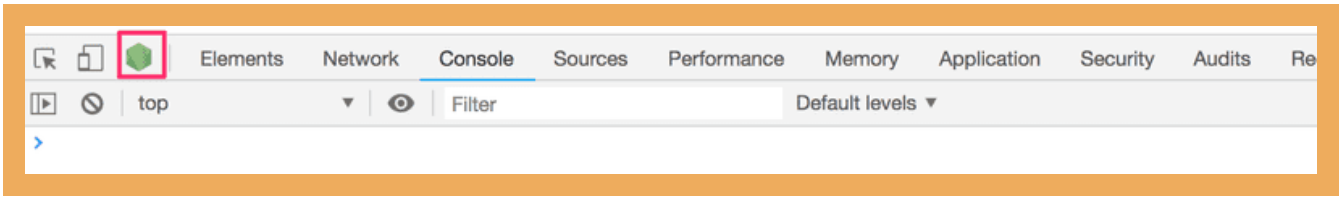
copy

You can also pass the `--inspect` flag to `nodemon`:

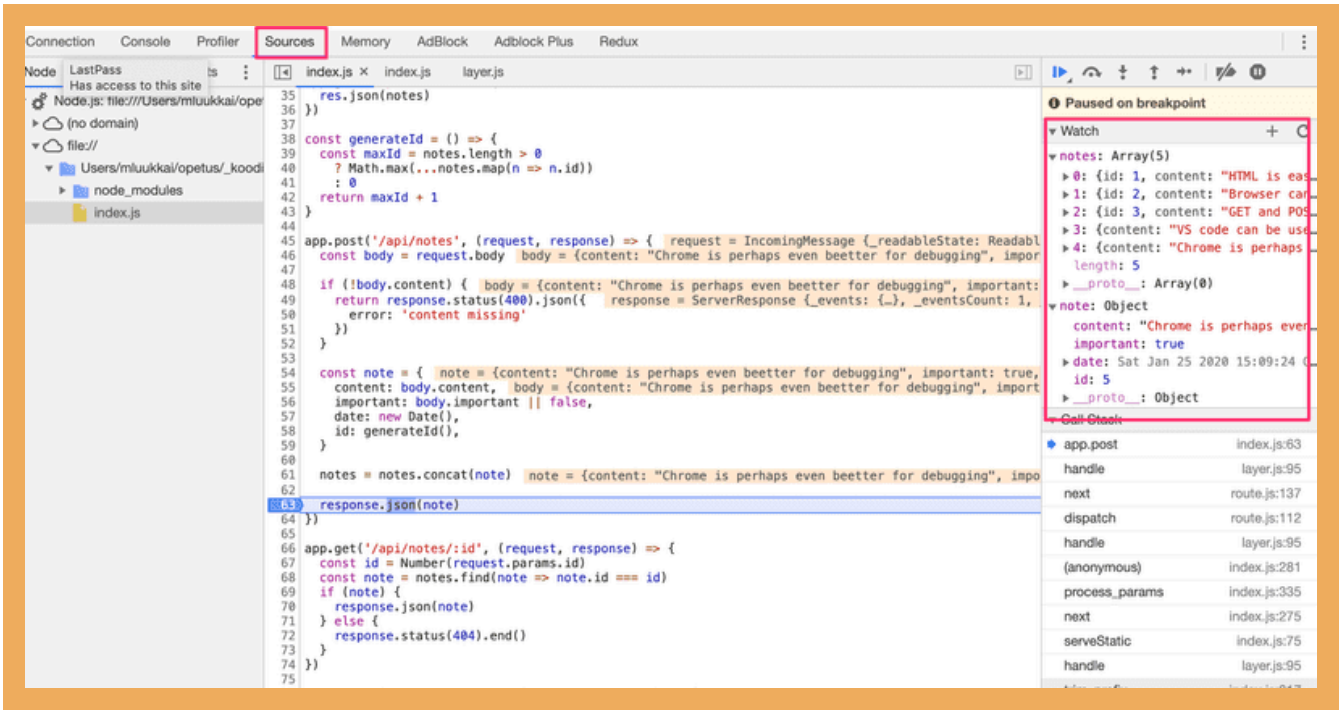
```
nodemon --inspect index.js
```

copy

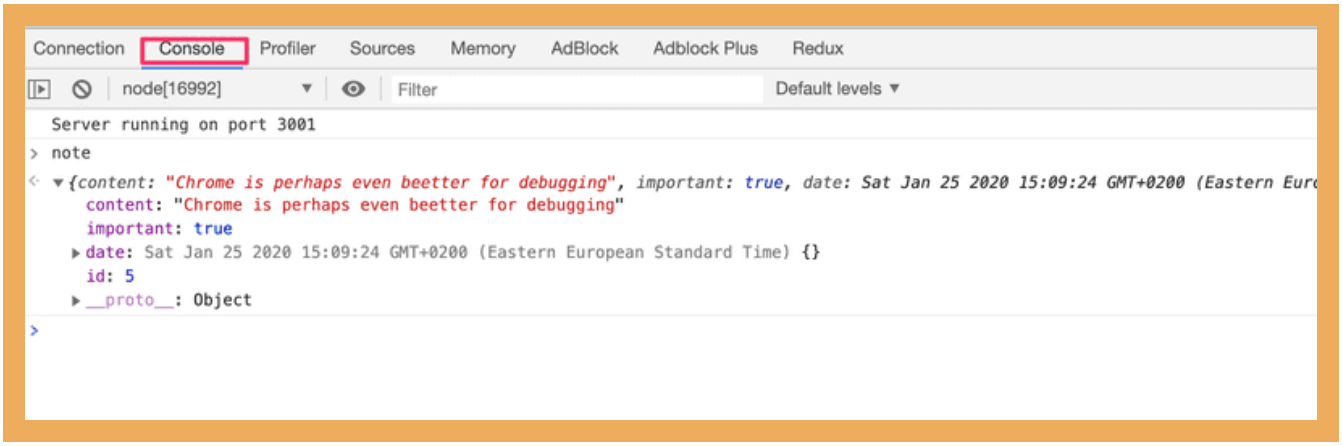
You can access the debugger by clicking the green icon - the node logo - that appears in the Chrome developer console:



The debugging view works the same way as it did with React applications. The *Sources* tab can be used for setting breakpoints where the execution of the code will be paused.



All of the application's *console.log* messages will appear in the *Console* tab of the debugger. You can also inspect values of variables and execute your own JavaScript code.



Question everything

Debugging Full Stack applications may seem tricky at first. Soon our application will also have a database in addition to the frontend and backend, and there will be many potential areas for bugs in the application.

When the application "does not work", we have to first figure out where the problem actually occurs. It's very common for the problem to exist in a place where you didn't expect it, and it can take minutes, hours, or even days before you find the source of the problem.

The key is to be systematic. Since the problem can exist anywhere, *you must question everything*, and eliminate all possibilities one by one. Logging to the console, Postman, debuggers, and experience will help.

When bugs occur, *the worst of all possible strategies* is to continue writing code. It will guarantee that your code will soon have even more bugs, and debugging them will be even more difficult. The Jidoka (stop and fix) principle from Toyota Production Systems is very effective in this situation as well.

MongoDB

To store our saved notes indefinitely, we need a database. Most of the courses taught at the University of Helsinki use relational databases. In most parts of this course, we will use MongoDB which is a document database.

The reason for using Mongo as the database is its lower complexity compared to a relational database. Part 13 of the course shows how to build Node.js backends that use a relational database.

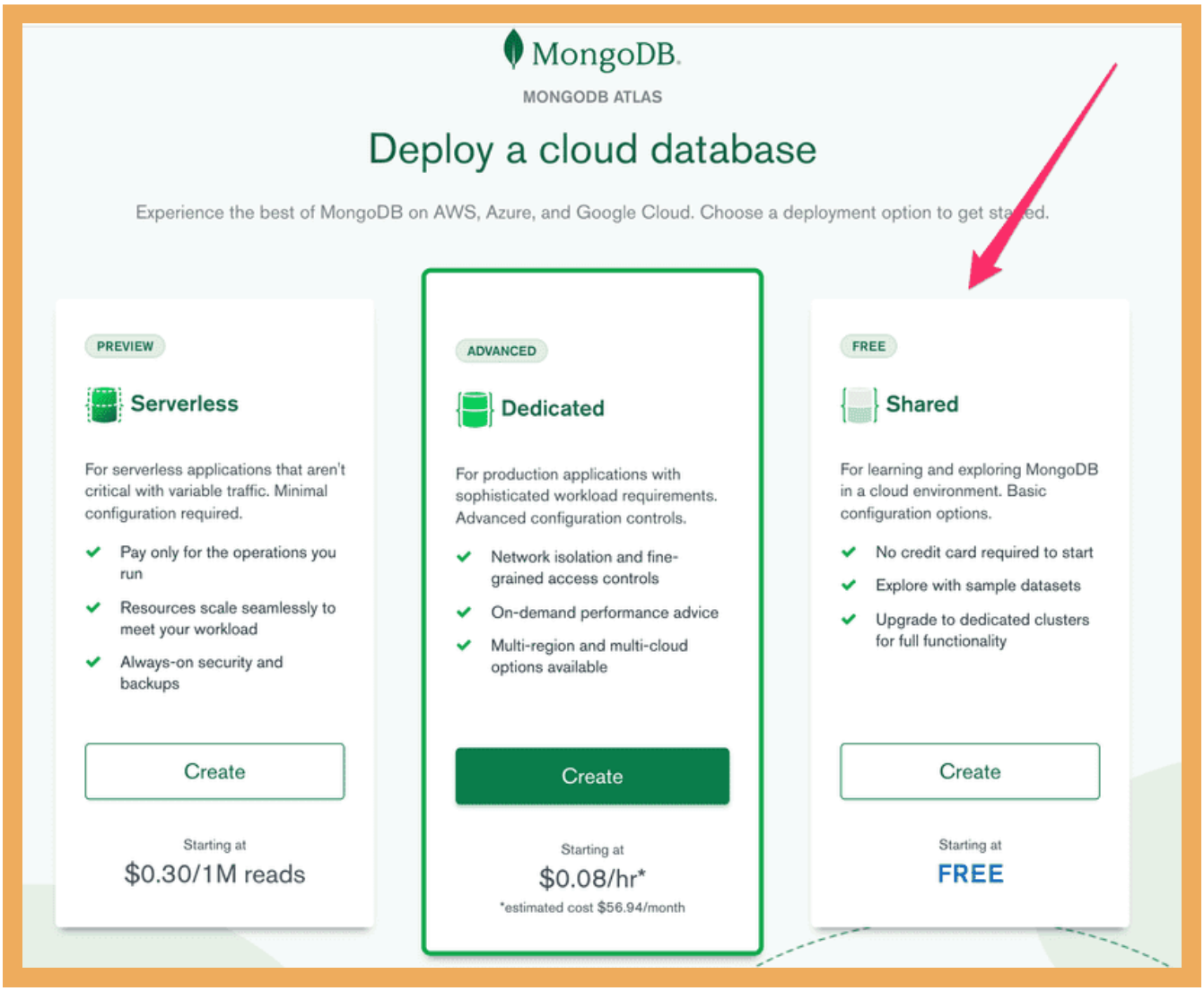
Document databases differ from relational databases in how they organize data as well as in the query languages they support. Document databases are usually categorized under the NoSQL umbrella term.

You can read more about document databases and NoSQL from the course material for week 7 of the Introduction to Databases course. Unfortunately, the material is currently only available in Finnish.

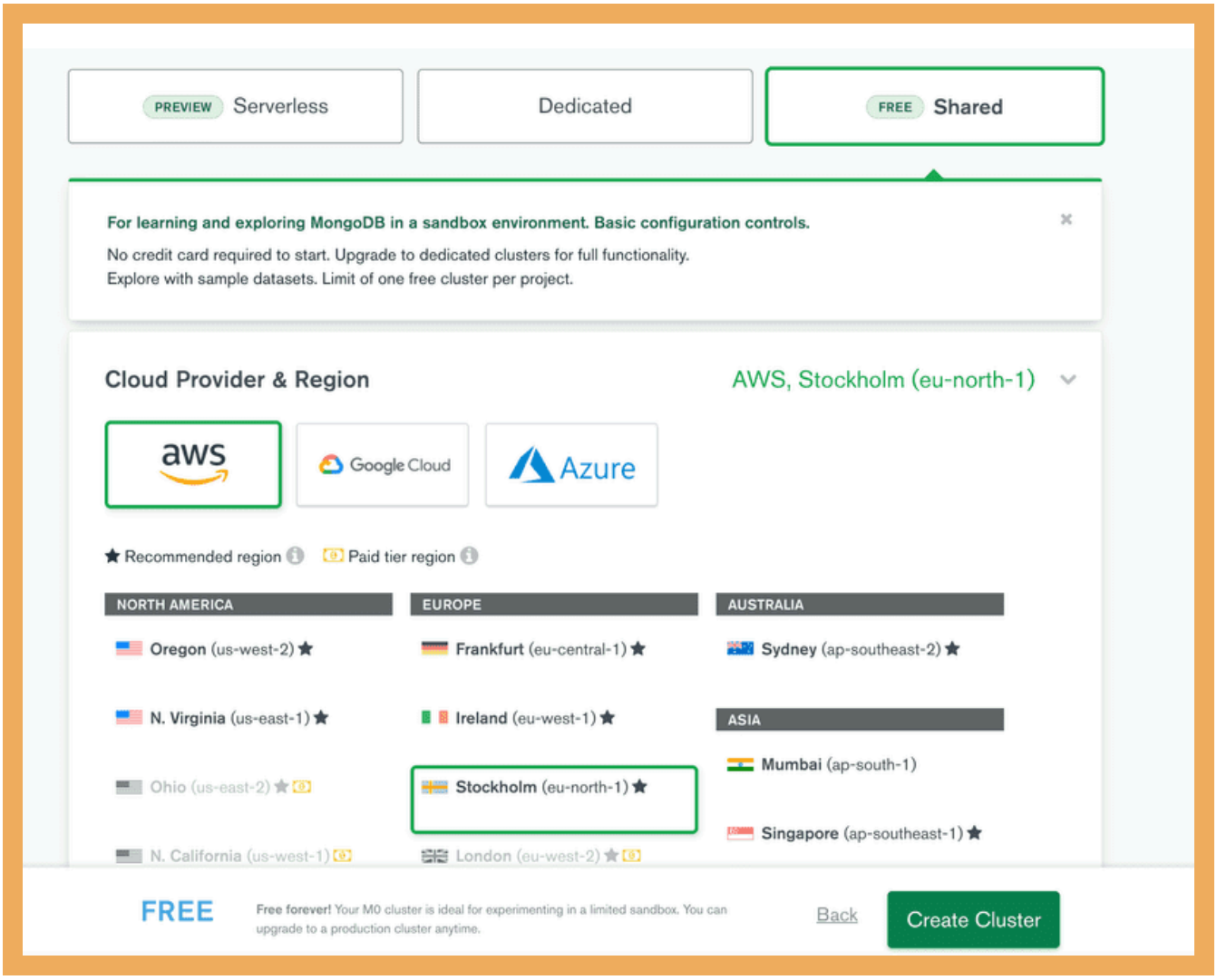
Read now the chapters on collections and documents from the MongoDB manual to get a basic idea of how a document database stores data.

Naturally, you can install and run MongoDB on your computer. However, the internet is also full of Mongo database services that you can use. Our preferred MongoDB provider in this course will be MongoDB Atlas.

Once you've created and logged into your account, let us start by selecting the free option:



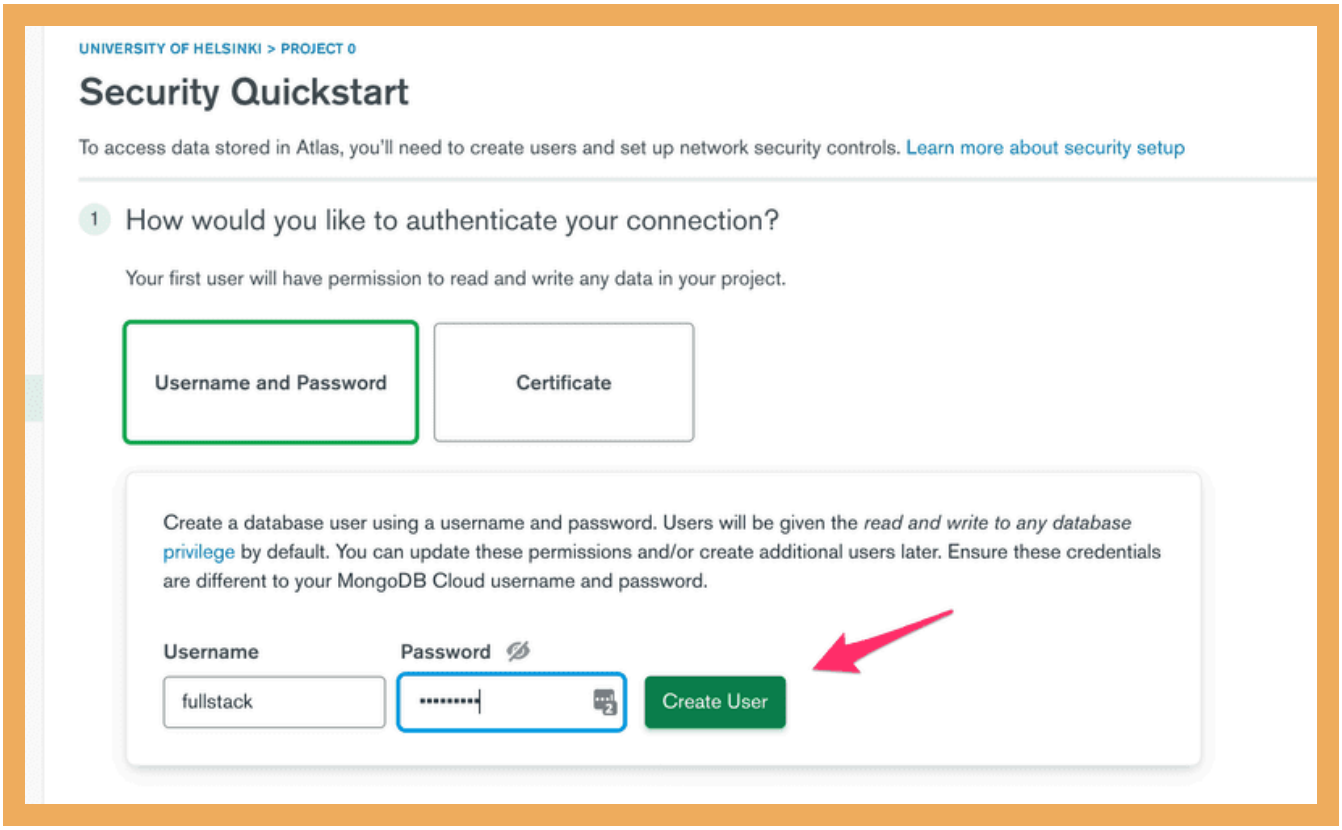
Pick the cloud provider and location and create the cluster:



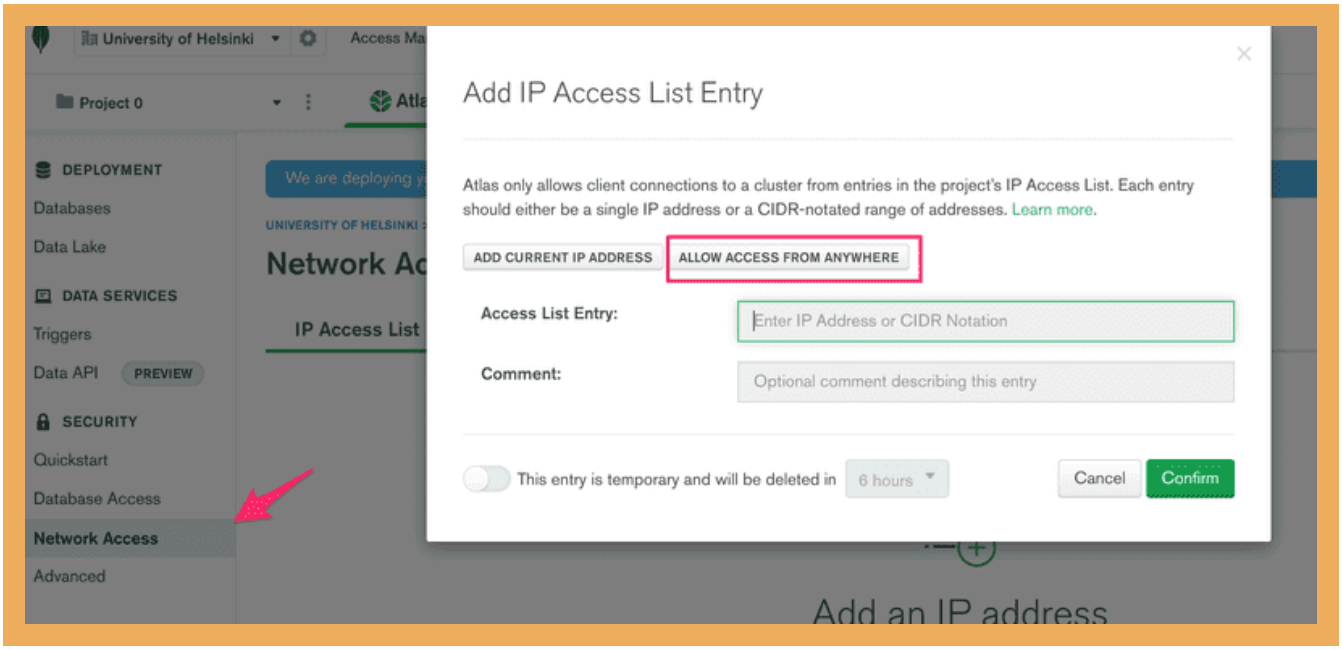
Let's wait for the cluster to be ready for use. This can take some minutes.

NB do not continue before the cluster is ready.

Let's use the *security* tab for creating user credentials for the database. Please note that these are not the same credentials you use for logging into MongoDB Atlas. These will be used for your application to connect to the database.

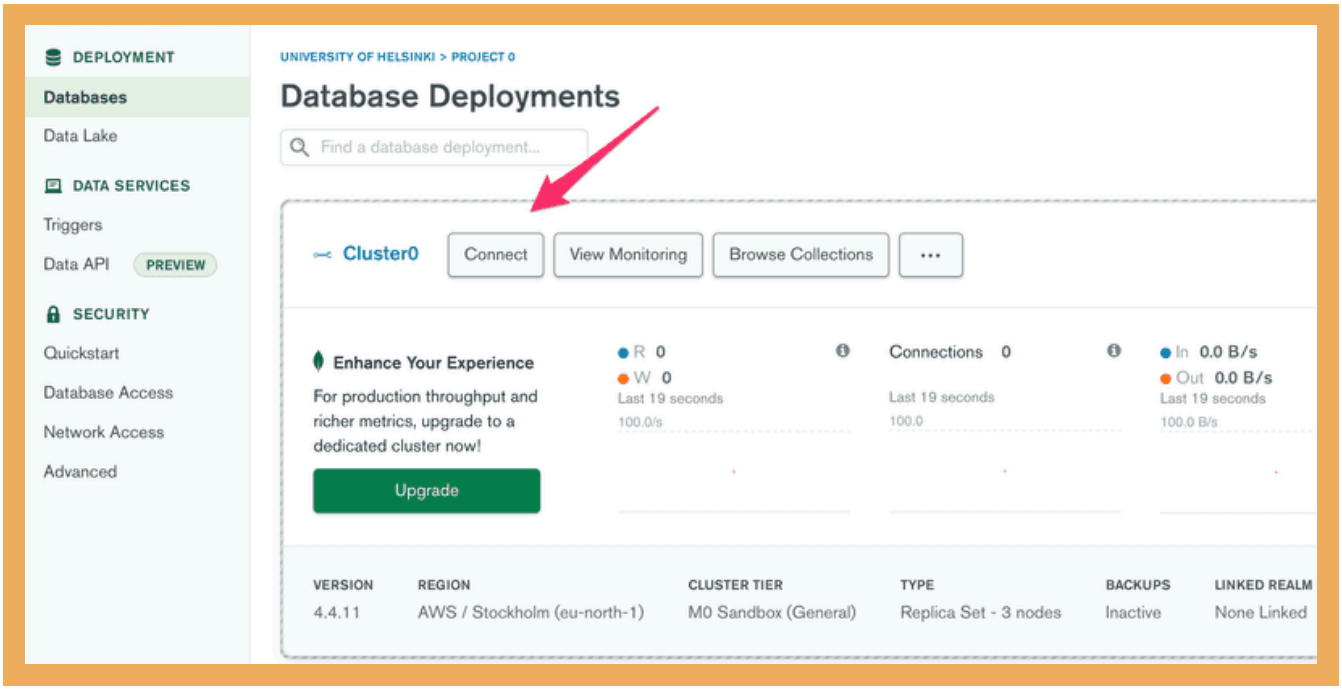


Next, we have to define the IP addresses that are allowed access to the database. For the sake of simplicity we will allow access from all IP addresses:

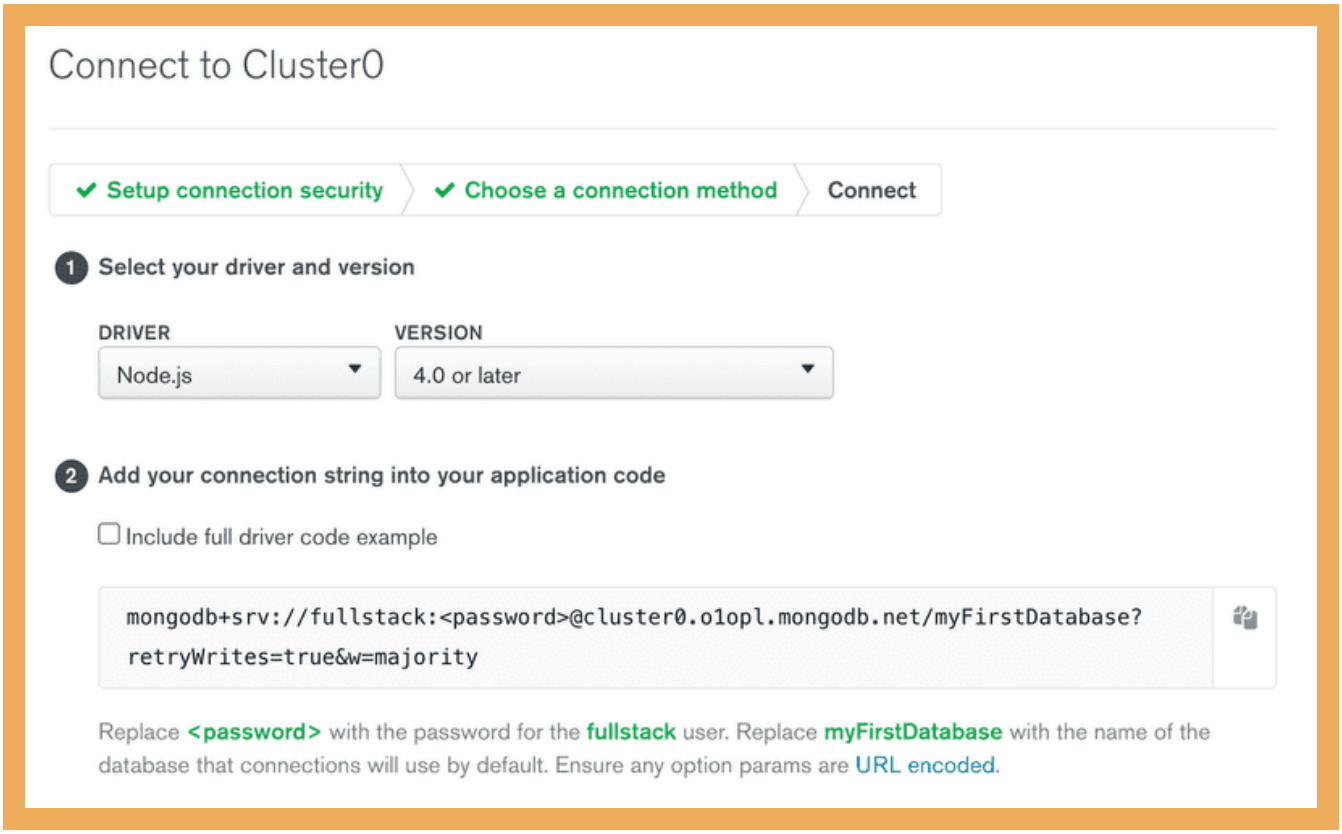


Note: In case the modal menu is different for you, according to MongoDB documentation, adding 0.0.0.0 as an IP allows access from anywhere as well.

Finally, we are ready to connect to our database. Start by clicking *connect*:



and choose: *Connect to your application*:



The view displays the *MongoDB URI*, which is the address of the database that we will supply to the MongoDB client library we will add to our application.

The address looks like this:

```
mongodb+srv://fullstack:thepasswordishere@cluster0.o1opl.mongodb.net/?
retryWrites=true&w=majority
```

copy

We are now ready to use the database.

We could use the database directly from our JavaScript code with the official MongoDB Node.js driver library, but it is quite cumbersome to use. We will instead use the Mongoose library that offers a higher-level API.

Mongoose could be described as an *object document mapper* (ODM), and saving JavaScript objects as Mongo documents is straightforward with this library.

Let's install Mongoose in our notes project backend:

```
npm install mongoose
```

copy

Let's not add any code dealing with Mongo to our backend just yet. Instead, let's make a practice application by creating a new file, *mongo.js* in the root of the notes backend application:

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

if (process.argv.length<3) {
  console.log('give password as argument')
  process.exit(1)
}

const password = process.argv[2]

const url =
  `mongodb+srv://fullstack:${password}@cluster0.o1opl.mongodb.net/?
retryWrites=true&w=majority`

mongoose.set('strictQuery',false)

mongoose.connect(url)

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

const Note = mongoose.model('Note', noteSchema)

const note = new Note({
  content: 'HTML is easy',
  important: true,
})
```

copy

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

NB: Depending on which region you selected when building your cluster, the *MongoDB URI* may be different from the example provided above. You should verify and use the correct URI that was generated from MongoDB Atlas.

The code also assumes that it will be passed the password from the credentials we created in MongoDB Atlas, as a command line parameter. We can access the command line parameter like this:

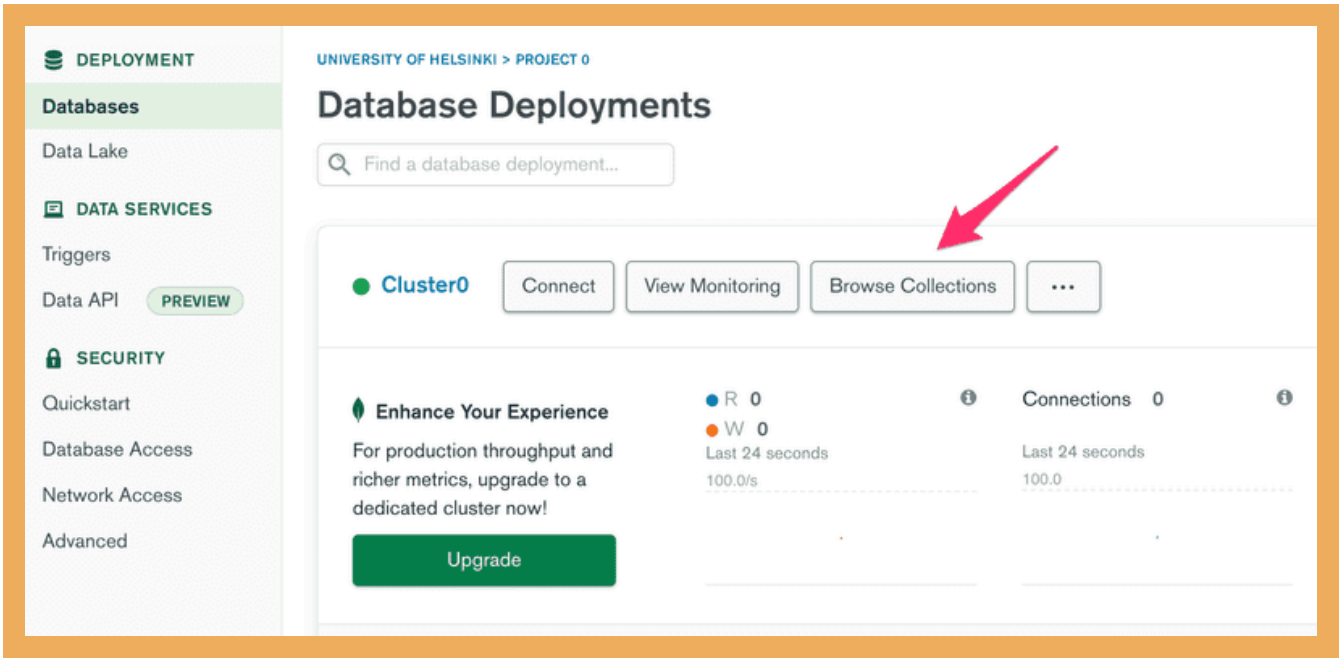
```
const password = process.argv[2]
```

copy

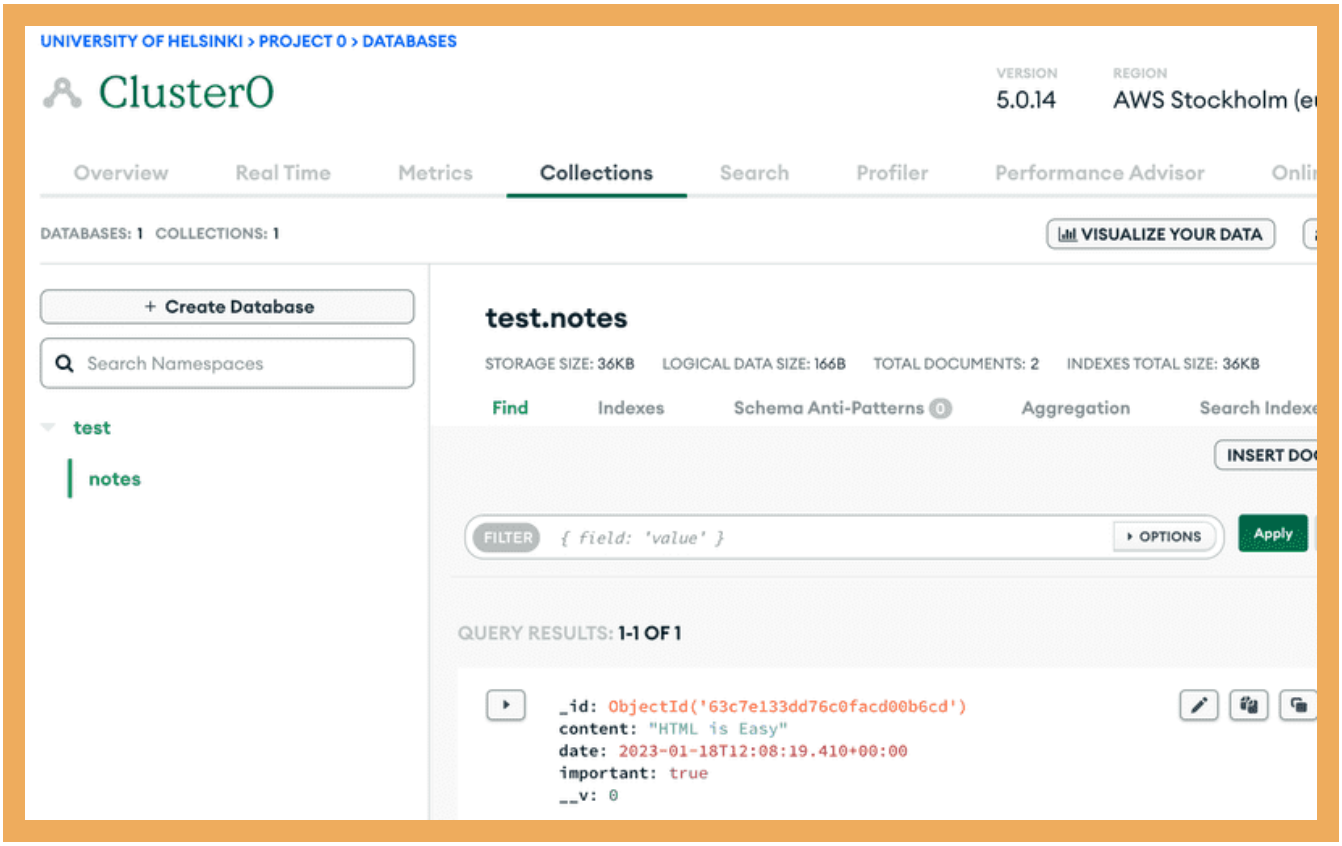
When the code is run with the command *node mongo.js yourPassword*, Mongo will add a new document to the database.

NB: Please note the password is the password created for the database user, not your MongoDB Atlas password. Also, if you created a password with special characters, then you'll need to URL encode that password.

We can view the current state of the database from the MongoDB Atlas from *Browse collections*, in the Database tab.



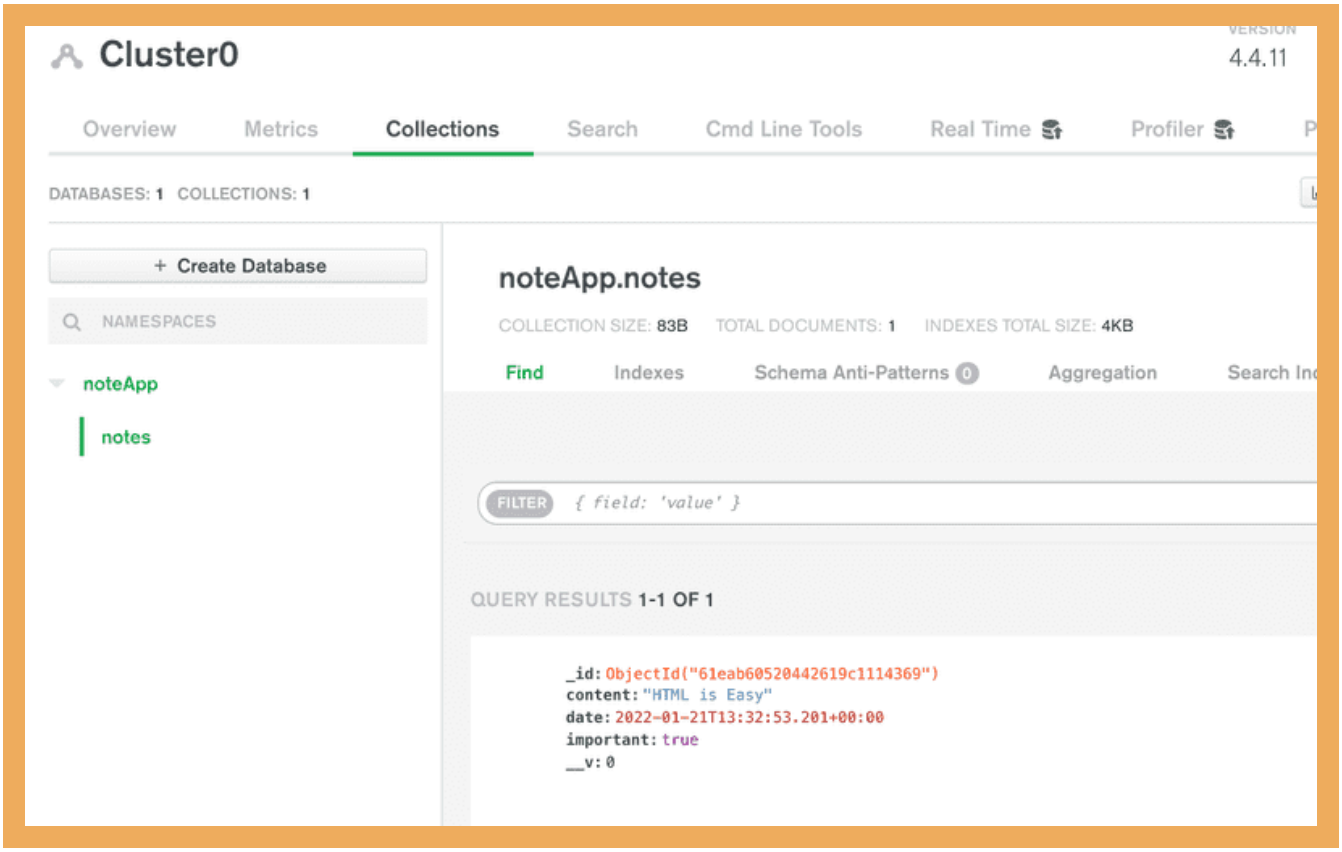
As the view states, the *document* matching the note has been added to the *notes* collection in the *myFirstDatabase* database.



Let's destroy the default database *test* and change the name of the database referenced in our connection string to *noteApp* instead, by modifying the URI:

```
const url =
  `mongodb+srv://fullstack:${password}@cluster0.o1opl.mongodb.net/noteApp?
retryWrites=true&w=majority`
```

Let's run our code again:



The data is now stored in the right database. The view also offers the *create database* functionality, that can be used to create new databases from the website. Creating a database like this is not necessary, since MongoDB Atlas automatically creates a new database when an application tries to connect to a database that does not exist yet.

Schema

After establishing the connection to the database, we define the schema for a note and the matching model:

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

const Note = mongoose.model('Note', noteSchema)
```

First, we define the schema of a note that is stored in the `noteSchema` variable. The schema tells Mongoose how the note objects are to be stored in the database.

In the `Note` model definition, the first *"Note"* parameter is the singular name of the model. The name of the collection will be the lowercase plural *notes*, because the Mongoose convention is to automatically name collections as the plural (e.g. *notes*) when the schema refers to them in the singular (e.g. *Note*).

Document databases like Mongo are *schemaless*, meaning that the database itself does not care about the structure of the data that is stored in the database. It is possible to store documents with completely different fields in the same collection.

The idea behind Mongoose is that the data stored in the database is given a *schema at the level of the application* that defines the shape of the documents stored in any given collection.

Creating and saving objects

Next, the application creates a new note object with the help of the *Note* model:

```
const note = new Note({
  content: 'HTML is Easy',
  important: false,
})
```

Models are *constructor functions* that create new JavaScript objects based on the provided parameters. Since the objects are created with the model's constructor function, they have all the properties of the model, which include methods for saving the object to the database.

Saving the object to the database happens with the appropriately named `save` method, which can be provided with an event handler with the `then` method:

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

When the object is saved to the database, the event handler provided to `then` gets called. The event handler closes the database connection with the command `mongoose.connection.close()`. If the connection is not closed, the program will never finish its execution.

The result of the save operation is in the `result` parameter of the event handler. The result is not that interesting when we're storing one object in the database. You can print the object to the console if you want to take a closer look at it while implementing your application or during debugging.

Let's also save a few more notes by modifying the data in the code and by executing the program again.

NB: Unfortunately the Mongoose documentation is not very consistent, with parts of it using callbacks in its examples and other parts, other styles, so it is not recommended to copy and paste code directly from there. Mixing promises with old-school callbacks in the same code is not recommended.

Fetching objects from the database

Let's comment out the code for generating new notes and replace it with the following:

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

When the code is executed, the program prints all the notes stored in the database:

```
→  backend git:(part3-4) ✗ node mongo.js
{
  _id: new ObjectId("63c7e133dd76c0facd00b6cd"),
  content: 'HTML is Easy',
  date: 2023-01-18T12:08:19.410Z,
  important: true,
  __v: 0
}
{
  _id: new ObjectId("63c7e781fdb247814b37f88a"),
  content: 'CSS is hard',
  date: 2023-01-18T12:35:13.345Z,
  important: true,
  __v: 0
}
{
  _id: new ObjectId("63c7e78d9e87b60dc6a713c9"),
  content: 'Mongoose makes things easy',
  date: 2023-01-18T12:35:25.475Z,
  important: true,
  __v: 0
}
```

The objects are retrieved from the database with the find method of the `Note` model. The parameter of the method is an object expressing search conditions. Since the parameter is an empty object `{}`, we get all of the notes stored in the `notes` collection.

The search conditions adhere to the Mongo search query syntax.

We could restrict our search to only include important notes like this:

```
Note.find({ important: true }).then(result => {
  // ...
```

```
})
```

Exercise 3.12.

3.12: Command-line database

Create a cloud-based MongoDB database for the phonebook application with MongoDB Atlas.

Create a *mongo.js* file in the project directory, that can be used for adding entries to the phonebook, and for listing all of the existing entries in the phonebook.

NB: Do not include the password in the file that you commit and push to GitHub!

The application should work as follows. You use the program by passing three command-line arguments (the first is the password), e.g.:

```
node mongo.js yourpassword Anna 040-1234556
```

copy

As a result, the application will print:

```
added Anna number 040-1234556 to phonebook
```

copy

The new entry to the phonebook will be saved to the database. Notice that if the name contains whitespace characters, it must be enclosed in quotes:

```
node mongo.js yourpassword "Arto Vihavainen" 045-1232456
```

copy

If the password is the only parameter given to the program, meaning that it is invoked like this:

```
node mongo.js yourpassword
```

copy

Then the program should display all of the entries in the phonebook:

```
phonebook:
Anna 040-1234556
```

copy

```
Arto Vihavainen 045-1232456
Ada Lovelace 040-1231236
```

You can get the command-line parameters from the `process.argv` variable.

NB: do not close the connection in the wrong place. E.g. the following code will not work:

```
Person
  .find({})
  .then(persons=> {
    // ...
  })

mongoose.connection.close()
```

In the code above the *mongoose.connection.close()* command will get executed immediately after the *Person.find* operation is started. This means that the database connection will be closed immediately, and the execution will never get to the point where *Person.find* operation finishes and the *callback* function gets called.

The correct place for closing the database connection is at the end of the callback function:

```
Person
  .find({})
  .then(persons=> {
    // ...
    mongoose.connection.close()
  })
```

NB: If you define a model with the name *Person*, mongoose will automatically name the associated collection as *people*.

Connecting the backend to a database

Now we have enough knowledge to start using Mongo in our notes application backend.

Let's get a quick start by copy-pasting the Mongoose definitions to the *index.js* file:

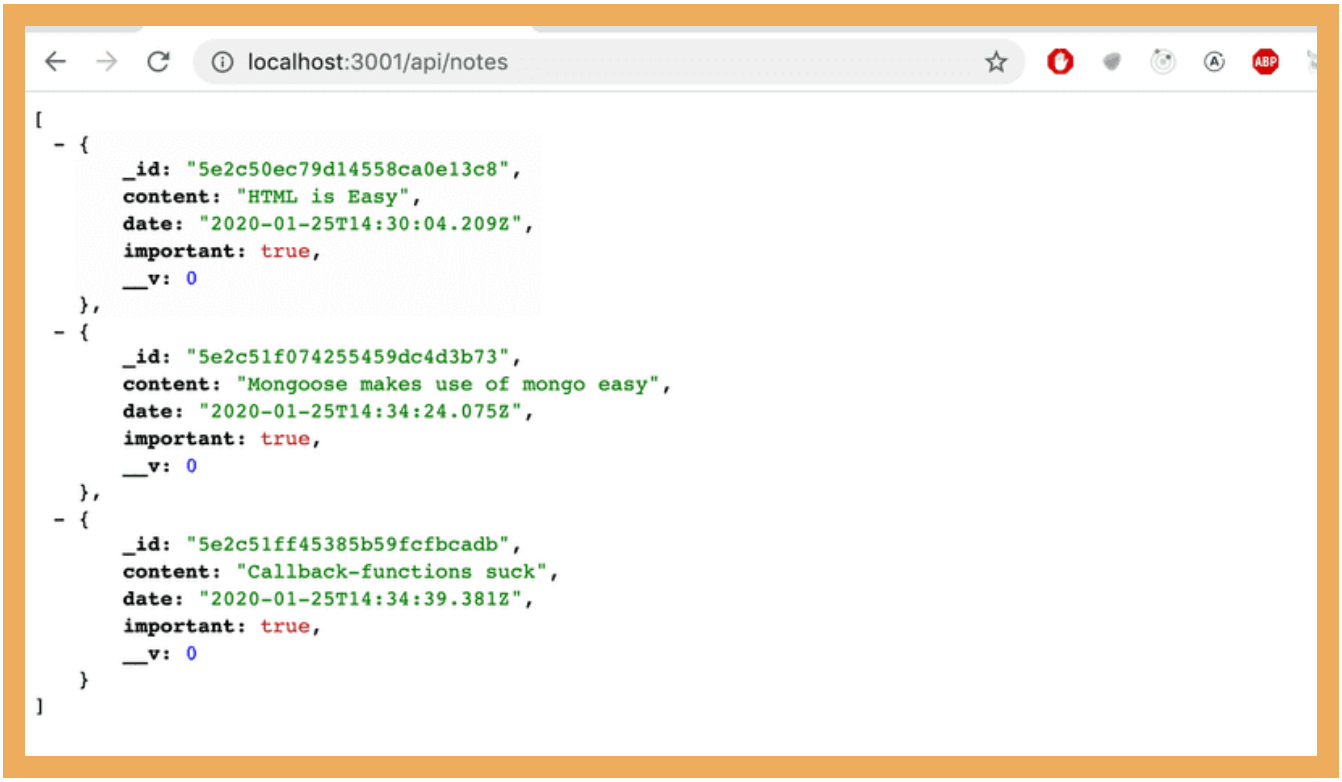
```
const mongoose = require('mongoose')

const password = process.argv[2]

// DO NOT SAVE YOUR PASSWORD TO GITHUB!!
```

```
const url =
  `mongodb+srv://fullstack:${password}@cluster0.o1opl.mongodb.net/?
retryWrites=true&w=majority`

mongoose.set('strictQuery',false)
mongoose.connect(url)

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

const Note = mongoose.model('Note', noteSchema)
```

Let's change the handler for fetching all notes to the following form:

```
app.get('/api/notes', (request, response) => {
  Note.find({}).then(notes => {
    response.json(notes)
  })
})
```

We can verify in the browser that the backend works for displaying all of the documents:



The application works almost perfectly. The frontend assumes that every object has a unique id in the *id* field. We also don't want to return the mongo versioning field *__v* to the frontend.

One way to format the objects returned by Mongoose is to modify the `toJSON` method of the schema, which is used on all instances of the models produced with that schema.

To modify the method we need to change the configurable options of the schema, options can be changed using the set method of the schema, see here for more info on this method: https://mongoosejs.com/docs/guide.html#options . See https://mongoosejs.com/docs/guide.html#toJSON and https://mongoosejs.com/docs/api.html#document_Document-toObject for more info on the `toJSON` option.

see https://mongoosejs.com/docs/api/document.html#transform for more info on the `transform` function.

```
noteSchema.set('toJSON', {
  transform: (document, returnedObject) => {
    returnedObject.id = returnedObject._id.toString()
    delete returnedObject._id
    delete returnedObject.__v
  }
})
```

Even though the *_id* property of Mongoose objects looks like a string, it is in fact an object. The `toJSON` method we defined transforms it into a string just to be safe. If we didn't make this change, it would cause more harm to us in the future once we start writing tests.

No changes are needed in the handler:

```
app.get('/api/notes', (request, response) => {
  Note.find({}).then(notes => {
    response.json(notes)
  })
})
```

The code automatically uses the defined `toJSON` when formatting notes to the response.

Moving db configuration to its own module

Before we refactor the rest of the backend to use the database, let's extract the Mongoose-specific code into its own module.

Let's create a new directory for the module called *models*, and add a file called *note.js*:

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

```
mongoose.set('strictQuery', false)

const url = process.env.MONGODB_URI

console.log('connecting to', url)

mongoose.connect(url)
  .then(result => {
    console.log('connected to MongoDB')
  })
  .catch(error => {
    console.log('error connecting to MongoDB:', error.message)
  })

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

noteSchema.set('toJSON', {
  transform: (document, returnedObject) => {
    returnedObject.id = returnedObject._id.toString()
    delete returnedObject._id
    delete returnedObject.__v
  }
})

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

Defining Node modules differs slightly from the way of defining ES6 modules in part 2.

The public interface of the module is defined by setting a value to the `module.exports` variable. We will set the value to be the *Note* model. The other things defined inside of the module, like the variables `mongoose` and `url` will not be accessible or visible to users of the module.

Importing the module happens by adding the following line to *index.js*:

```
const Note = require('./models/note')
```

copy

This way the `Note` variable will be assigned to the same object that the module defines.

The way that the connection is made has changed slightly:

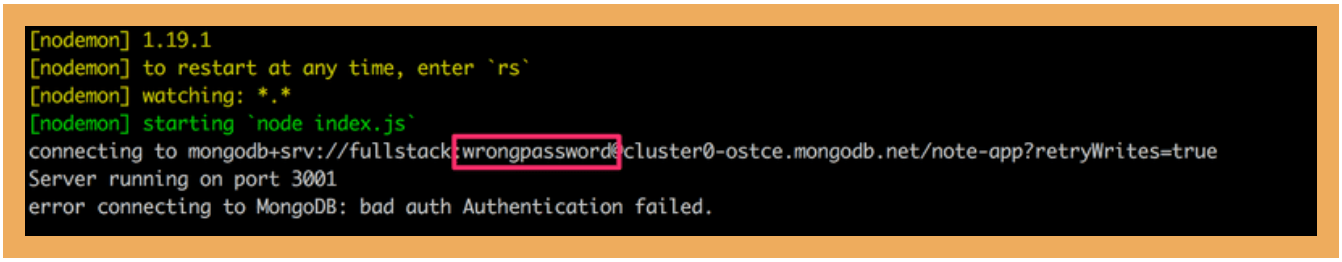
```
const url = process.env.MONGODB_URI

console.log('connecting to', url)

mongoose.connect(url)
  .then(result => {
    console.log('connected to MongoDB')
```

copy

```
})
.catch(error => {
  console.log('error connecting to MongoDB:', error.message)
})
```

It's not a good idea to hardcode the address of the database into the code, so instead the address of the database is passed to the application via the `MONGODB_URI` environment variable.

The method for establishing the connection is now given functions for dealing with a successful and unsuccessful connection attempt. Both functions just log a message to the console about the success status:



There are many ways to define the value of an environment variable. One way would be to define it when the application is started:

```
MONGODB_URI=address_here npm run dev
```

copy

A more sophisticated way is to use the dotenv library. You can install the library with the command:

```
npm install dotenv
```

copy

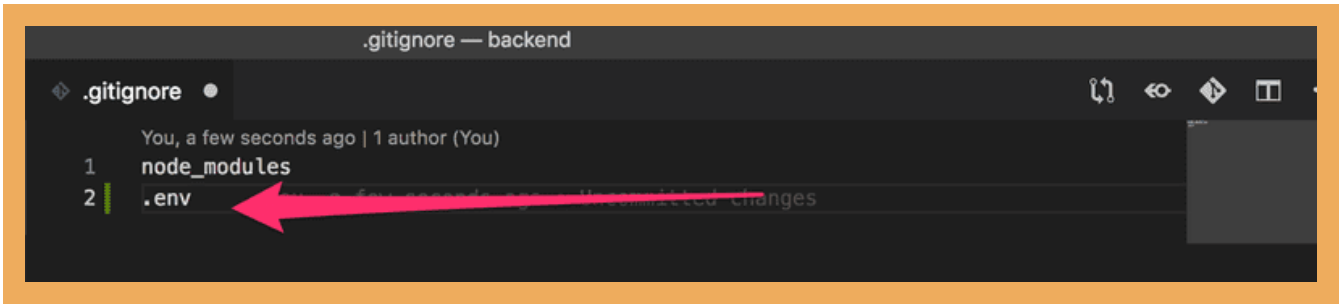
To use the library, we create a *.env* file at the root of the project. The environment variables are defined inside of the file, and it can look like this:

```
MONGODB_URI=mongodb+srv://fullstack:thepasswordishere@cluster0.o1opl.mongodb.net/noteApp?
retryWrites=true&w=majority
PORT=3001
```

copy

We also added the hardcoded port of the server into the `PORT` environment variable.

The *.env* file should be gitignored right away since we do not want to publish any confidential information publicly online!



The environment variables defined in the *.env* file can be taken into use with the expression `require('dotenv').config()` and you can reference them in your code just like you would reference normal environment variables, with the `process.env.MONGODB_URI` syntax.

Let's change the *index.js* file in the following way:

```
require('dotenv').config()
const express = require('express')
const app = express()
const Note = require('./models/note')

// ..

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

It's important that *dotenv* gets imported before the *note* model is imported. This ensures that the environment variables from the *.env* file are available globally before the code from the other modules is imported.

Important note to Fly.io users

Because GitHub is not used with Fly.io, the file .env also gets to the Fly.io servers when the app is deployed. Because of this, the env variables defined in the file will be available there.

However, a better option is to prevent .env from being copied to Fly.io by creating in the project root the file `.dockerignore`, with the following contents

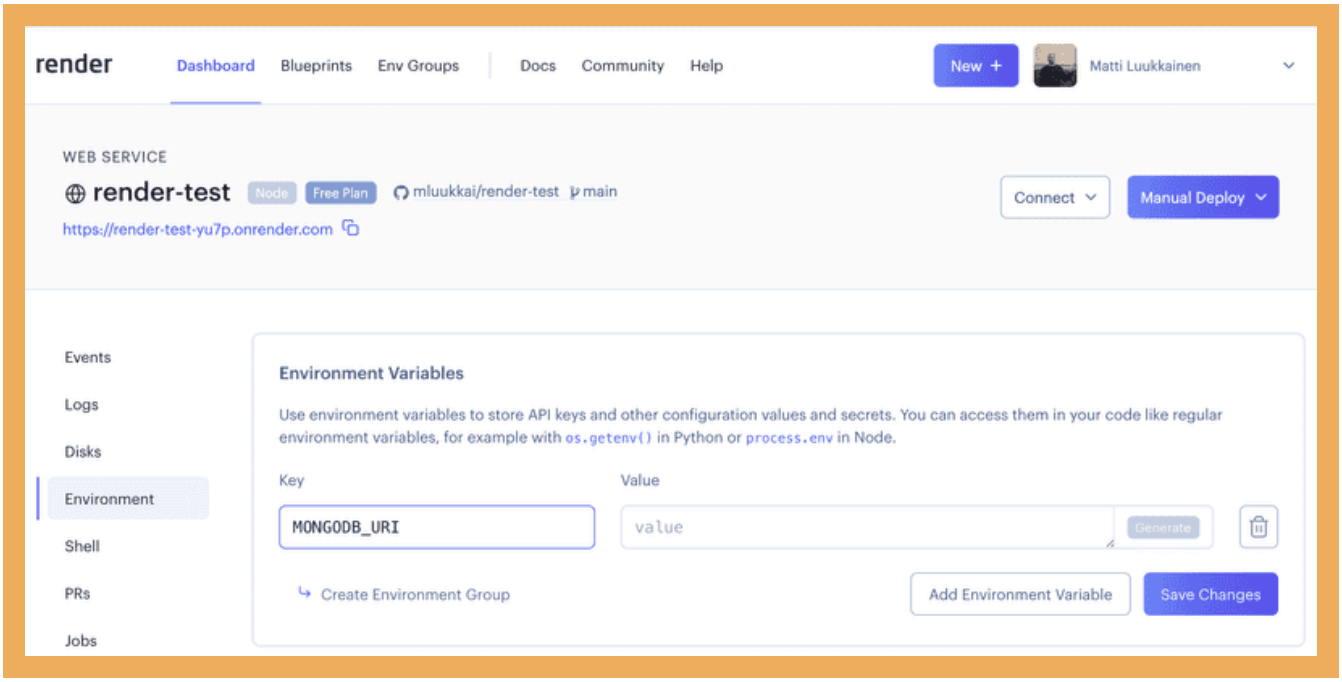
```
.env
```

and set the env value from the command line with the command:

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

Since the PORT also is defined in our .env it is actually essential to ignore the file in Fly.io since otherwise the app starts in the wrong port.

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



Set just the URL starting with *mongodb+srv://...* to the `value` field.

Using database in route handlers

Next, let's change the rest of the backend functionality to use the database.

Creating a new note is accomplished like this:

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

  if (body.content === undefined) {
    return response.status(400).json({ error: 'content missing' })
  }

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

```
  note.save().then(savedNote => {
    response.json(savedNote)
  })
})
```

The note objects are created with the `Note` constructor function. The response is sent inside of the callback function for the `save` operation. This ensures that the response is sent only if the operation succeeded. We will discuss error handling a little bit later.

The `savedNote` parameter in the callback function is the saved and newly created note. The data sent back in the response is the formatted version created automatically with the `toJSON` method:

```
response.json(savedNote)
```

copy

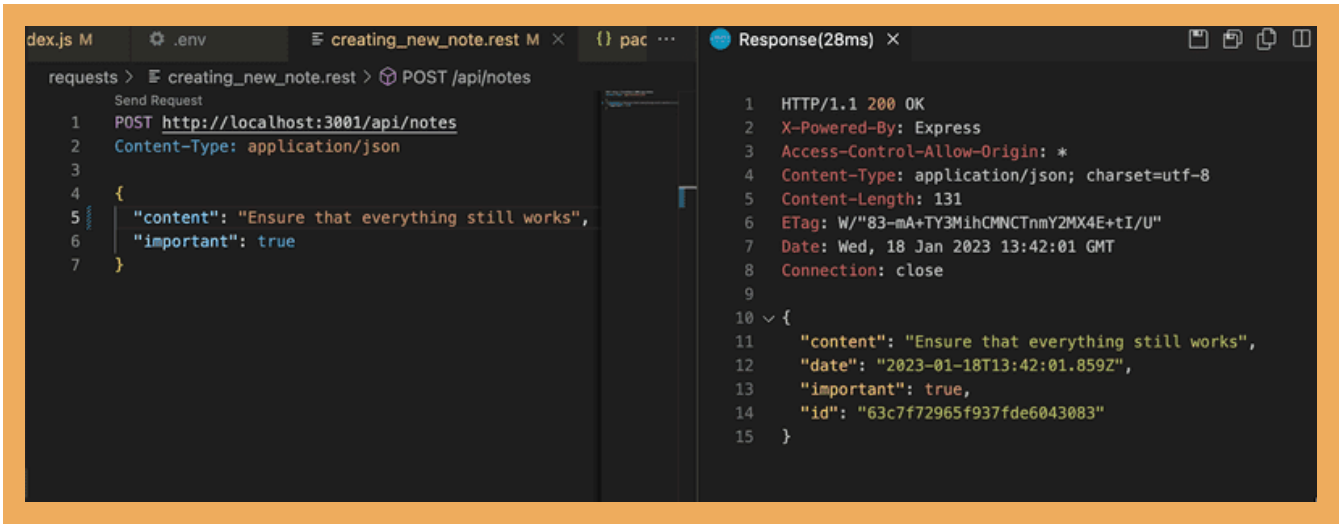
Using Mongoose's findById method, fetching an individual note gets changed into the following:

```
app.get('/api/notes/:id', (request, response) => {
  Note.findById(request.params.id).then(note => {
    response.json(note)
  })
})
```

copy

Verifying frontend and backend integration

When the backend gets expanded, it's a good idea to test the backend first with **the browser, Postman or the VS Code REST client**. Next, let's try creating a new note after taking the database into use:



Only once everything has been verified to work in the backend, is it a good idea to test that the frontend works with the backend. It is highly inefficient to test things exclusively through the frontend.

It's probably a good idea to integrate the frontend and backend one functionality at a time. First, we could implement fetching all of the notes from the database and test it through the backend endpoint in the browser. After this, we could verify that the frontend works with the new backend. Once everything seems to be working, we would move on to the next feature.

Once we introduce a database into the mix, it is useful to inspect the state persisted in the database, e.g. from the control panel in MongoDB Atlas. Quite often little Node helper programs like the *mongo.js* program we wrote earlier can be very helpful during development.

You can find the code for our current application in its entirety in the *part3-4* branch of this GitHub repository.

Exercises 3.13.-3.14.

The following exercises are pretty straightforward, but if your frontend stops working with the backend, then finding and fixing the bugs can be quite interesting.

3.13: Phonebook database, step 1

Change the fetching of all phonebook entries so that the data is *fetched from the database*.

Verify that the frontend works after the changes have been made.

In the following exercises, write all Mongoose-specific code into its own module, just like we did in the chapter Database configuration into its own module.

3.14: Phonebook database, step 2

Change the backend so that new numbers are *saved to the database*. Verify that your frontend still works after the changes.

At this stage, you can ignore whether there is already a person in the database with the same name as the person you are adding.

Error handling

If we try to visit the URL of a note with an id that does not exist e.g. http://localhost:3001/api/notes/5c41c90e84d891c15dfa3431 where *5c41c90e84d891c15dfa3431* is not an id stored in the database, then the response will be `null`.

Let's change this behavior so that if a note with the given id doesn't exist, the server will respond to the request with the HTTP status code 404 not found. In addition let's implement a simple `catch` block to handle cases where the promise returned by the `findById` method is *rejected*:

```
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(500).end()
    })
})
```

If no matching object is found in the database, the value of `note` will be `null` and the `else` block is executed. This results in a response with the status code *404 not found*. If a promise returned by the `findById` method is rejected, the response will have the status code *500 internal server error*. The console displays more detailed information about the error.

On top of the non-existing note, there's one more error situation that needs to be handled. In this situation, we are trying to fetch a note with the wrong kind of `id`, meaning an `id` that doesn't match the Mongo identifier format.

If we make the following request, we will get the error message shown below:

```
Method: GET
Path:   /api/notes/someInvalidId
Body:   {}
---
{ CastError: Cast to ObjectId failed for value "someInvalidId" at path "_id"
    at CastError (/Users/mluukkai/opetus/_fullstack/osa3-muisiinpanot/node_modules/mongoose/
    at ObjectId.cast (/Users/mluukkai/opetus/_fullstack/osa3-muisiinpanot/node_modules/mongo
    ...
```

Given a malformed id as an argument, the `findById` method will throw an error causing the returned promise to be rejected. This will cause the callback function defined in the `catch` block to be called.

Let's make some small adjustments to the response in the `catch` block:

```
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' })
    })
})
```

If the format of the id is incorrect, then we will end up in the error handler defined in the `catch` block. The appropriate status code for the situation is 400 Bad Request because the situation fits the description perfectly:

> *The 400 (Bad Request) status code indicates that the server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).*

We have also added some data to the response to shed some light on the cause of the error.

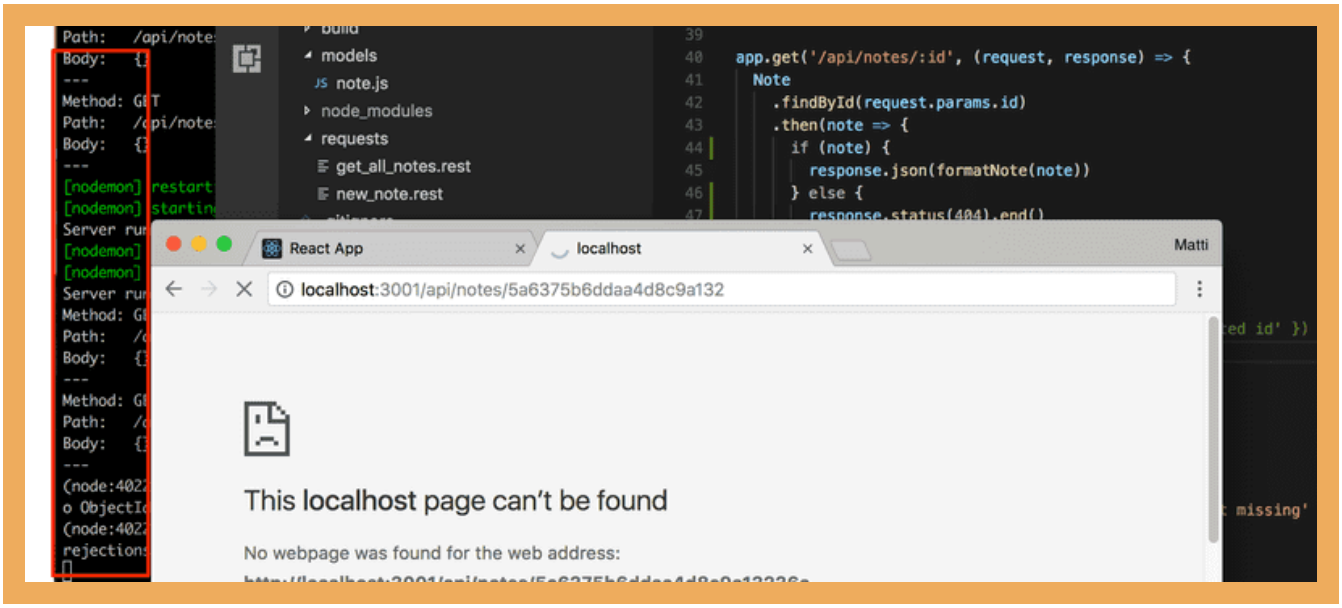
When dealing with Promises, it's almost always a good idea to add error and exception handling. Otherwise, you will find yourself dealing with strange bugs.

It's never a bad idea to print the object that caused the exception to the console in the error handler:

```
.catch(error => {
  console.log(error)
  response.status(400).send({ error: 'malformatted id' })
})
```

The reason the error handler gets called might be something completely different than what you had anticipated. If you log the error to the console, you may save yourself from long and frustrating debugging sessions. Moreover, most modern services where you deploy your application support some form of logging system that you can use to check these logs. As mentioned, Fly.io is one.

Every time you're working on a project with a backend, *it is critical to keep an eye on the console output of the backend*. If you are working on a small screen, it is enough to just see a tiny slice of the output in the background. Any error messages will catch your attention even when the console is far back in the background:



Moving error handling into middleware

We have written the code for the error handler among the rest of our code. This can be a reasonable solution at times, but there are cases where it is better to implement all error handling in a single place. This can be particularly useful if we want to report data related to errors to an external error-tracking system like Sentry later on.

Let's change the handler for the */api/notes/:id* route so that it passes the error forward with the `next` function. The next function is passed to the handler as the third parameter:

```
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))
})
```

The error that is passed forward is given to the `next` function as a parameter. If `next` was called without an argument, then the execution would simply move onto the next route or middleware. If the `next` function is called with an argument, then the execution will continue to the *error handler middleware*.

Express error handlers are middleware that are defined with a function that accepts *four parameters*. Our error handler looks like this:

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

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

  next(error)
}

// this has to be the last loaded middleware, also all the routes should be registered before this!
app.use(errorHandler)
```

The error handler checks if the error is a *CastError* exception, in which case we know that the error was caused by an invalid object id for Mongo. In this situation, the error handler will send a response to the browser with the response object passed as a parameter. In all other error situations, the middleware passes the error forward to the default Express error handler.

Note that the error-handling middleware has to be the last loaded middleware, also all the routes should be registered before the error-handler!

The order of middleware loading

The execution order of middleware is the same as the order that they are loaded into Express with the `app.use` function. For this reason, it is important to be careful when defining middleware.

The correct order is the following:

```
app.use(express.static('dist'))
app.use(express.json())
app.use(requestLogger)

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)
```

The json-parser middleware should be among the very first middleware loaded into Express. If the order was the following:

```
app.use(requestLogger) // request.body is undefined!

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

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

Then the JSON data sent with the HTTP requests would not be available for the logger middleware or the POST route handler, since the `request.body` would be `undefined` at that point.

It's also important that the middleware for handling unsupported routes is next to the last middleware that is loaded into Express, just before the error handler.

For example, the following loading order would cause an issue:

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

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

app.get('/api/notes', (request, response) => {
  // ...
})
```

Now the handling of unknown endpoints is ordered *before the HTTP request handler*. Since the unknown endpoint handler responds to all requests with *404 unknown endpoint*, no routes or middleware will be called after the response has been sent by unknown endpoint middleware. The only exception to this is the error handler which needs to come at the very end, after the unknown endpoints handler.

Other operations

Let's add some missing functionality to our application, including deleting and updating an individual note.

The easiest way to delete a note from the database is with the findByIdAndDelete method:

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

In both of the "successful" cases of deleting a resource, the backend responds with the status code *204 no content*. The two different cases are deleting a note that exists, and deleting a note that does not exist in the database. The `result` callback parameter could be used for checking if a resource was actually deleted, and we could use that information for returning different status codes for the two cases if we deem it necessary. Any exception that occurs is passed onto the error handler.

The toggling of the importance of a note can be easily accomplished with the findByIdAndUpdate method.

```
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))
})
```

In the code above, we also allow the content of the note to be edited.

Notice that the `findByIdAndUpdate` method receives a regular JavaScript object as its argument, and not a new note object created with the `Note` constructor function.

There is one important detail regarding the use of the `findByIdAndUpdate` method. By default, the `updatedNote` parameter of the event handler receives the original document without the modifications. We added the optional `{ new: true }` parameter, which will cause our event handler to be called with the new modified document instead of the original.

After testing the backend directly with Postman or the VS Code REST client, we can verify that it seems to work. The frontend also appears to work with the backend using the database.

You can find the code for our current application in its entirety in the *part3-5* branch of this GitHub repository.

A true full stack developer's oath

It is again time for the exercises. The complexity of our app has now taken another step since besides frontend and backend we also have a database. There are indeed really many potential sources of error.

So we should once more extend our oath:

Full stack development is *extremely hard*, that is why I will use all the possible means to make it easier

- I will have my browser developer console open all the time
- I will use the network tab of the browser dev tools to ensure that frontend and backend are communicating as I expect
- I will constantly keep an eye on the state of the server to make sure that the data sent there by the frontend is saved there as I expect
- *I will keep an eye on the database: does the backend save data there in the right format*
- I progress with small steps
- I will write lots of `console.log` statements to make sure I understand how the code behaves and to help pinpoint problems
- If my code does not work, I will not write more code. Instead, I start deleting the code until it works or just return to a state when everything was still working
- When I ask for help in the course Discord or Telegram channel or elsewhere I formulate my questions properly, see here how to ask for help

Exercises 3.15.-3.18.

3.15: Phonebook database, step 3

Change the backend so that deleting phonebook entries is reflected in the database.

Verify that the frontend still works after making the changes.

3.16: Phonebook database, step 4

Move the error handling of the application to a new error handler middleware.

3.17*: Phonebook database, step 5

If the user tries to create a new phonebook entry for a person whose name is already in the phonebook, the frontend will try to update the phone number of the existing entry by making an HTTP PUT request to the entry's unique URL.

Modify the backend to support this request.

Verify that the frontend works after making your changes.

3.18*: Phonebook database step 6

Also update the handling of the *api/persons/:id* and *info* routes to use the database, and verify that they work directly with the browser, Postman, or VS Code REST client.

Inspecting an individual phonebook entry from the browser should look like this:

Propose changes to material

Part 3b
Previous part

Part 3d
Next part

About course

Course contents

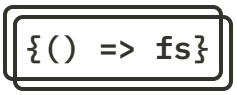
FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON



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
```

```
  },
  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.

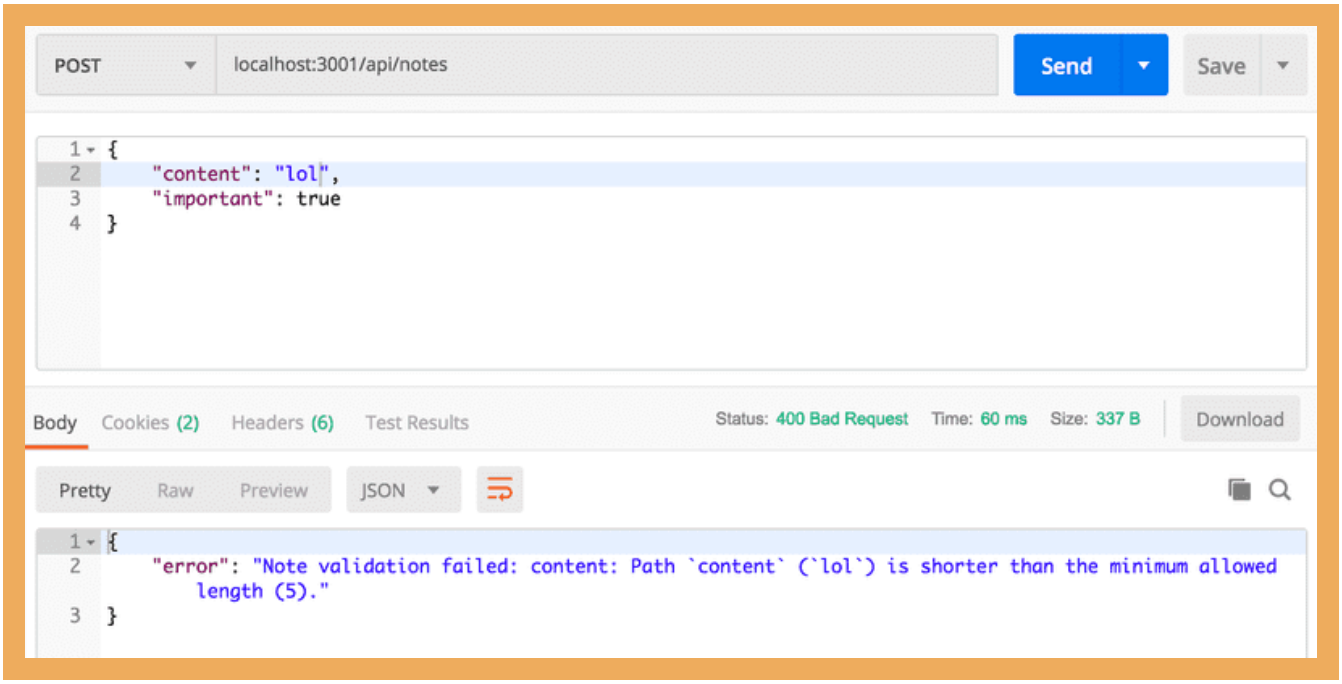
If we try to store an object in the database that breaks one of the constraints, the operation will throw an exception. Let's change our handler for creating a new note so that it passes any potential exceptions to the error handler middleware:

```
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.

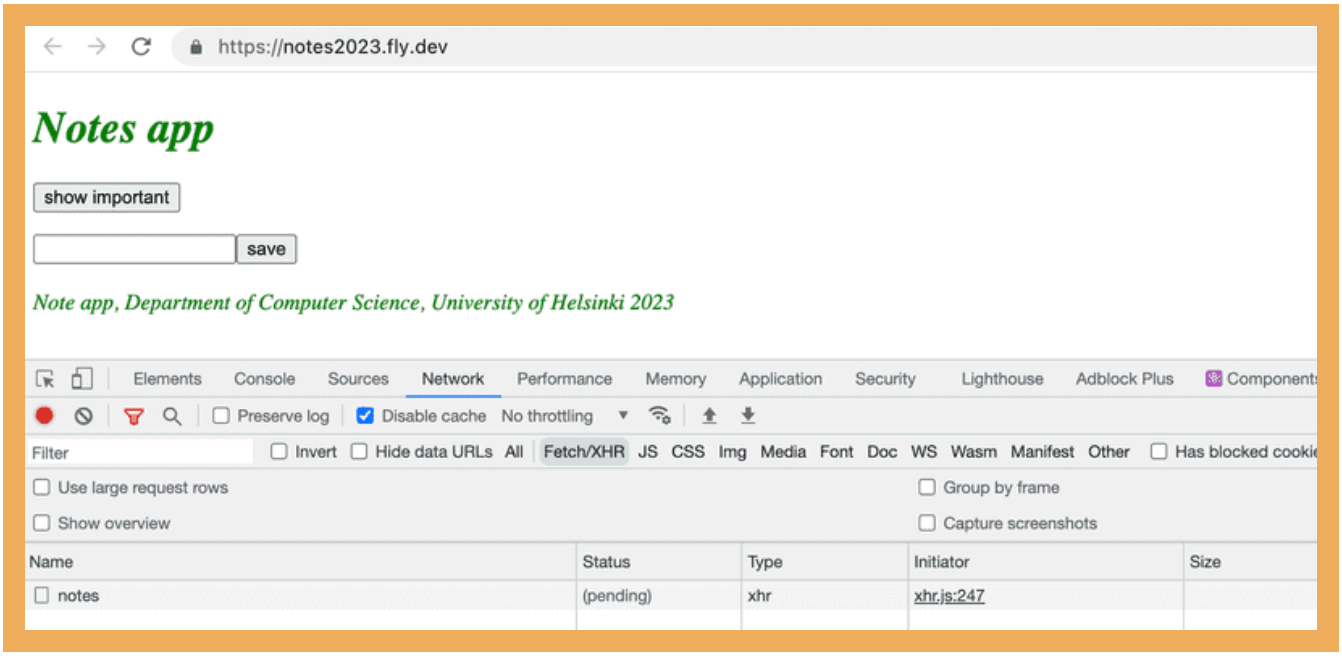
The environment variables defined in dotenv will only be used when the backend is not in *production mode*, i.e. Fly.io or Render.

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'
```

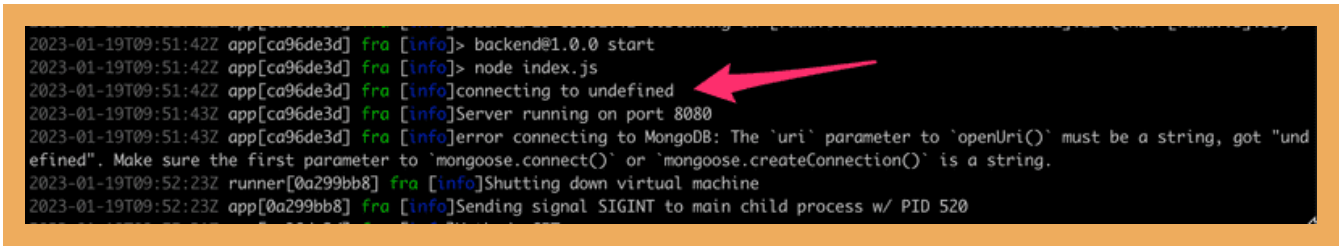
When the app is being developed, it is more than likely that something fails. Eg. when I deployed my app for the first time with the database, not a single note was seen:



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` :

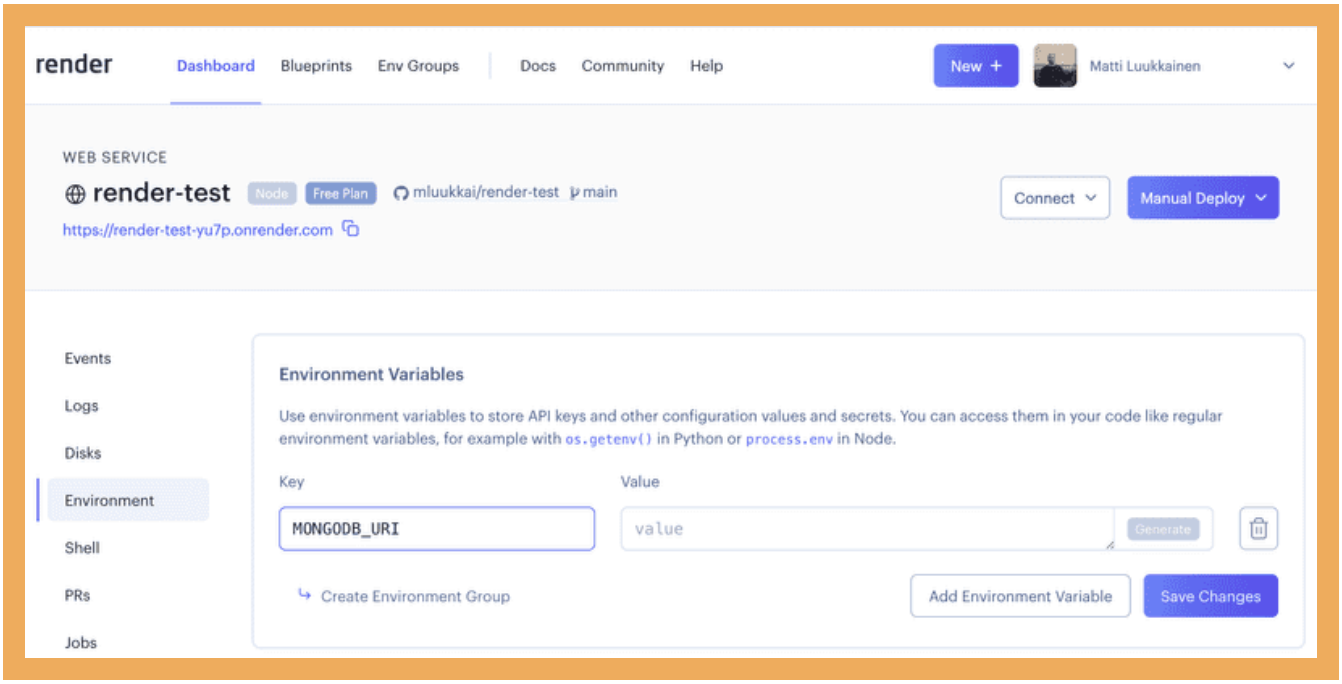


The database url was `undefined` , so the command `fly secrets set MONGODB_URI` was forgotten.

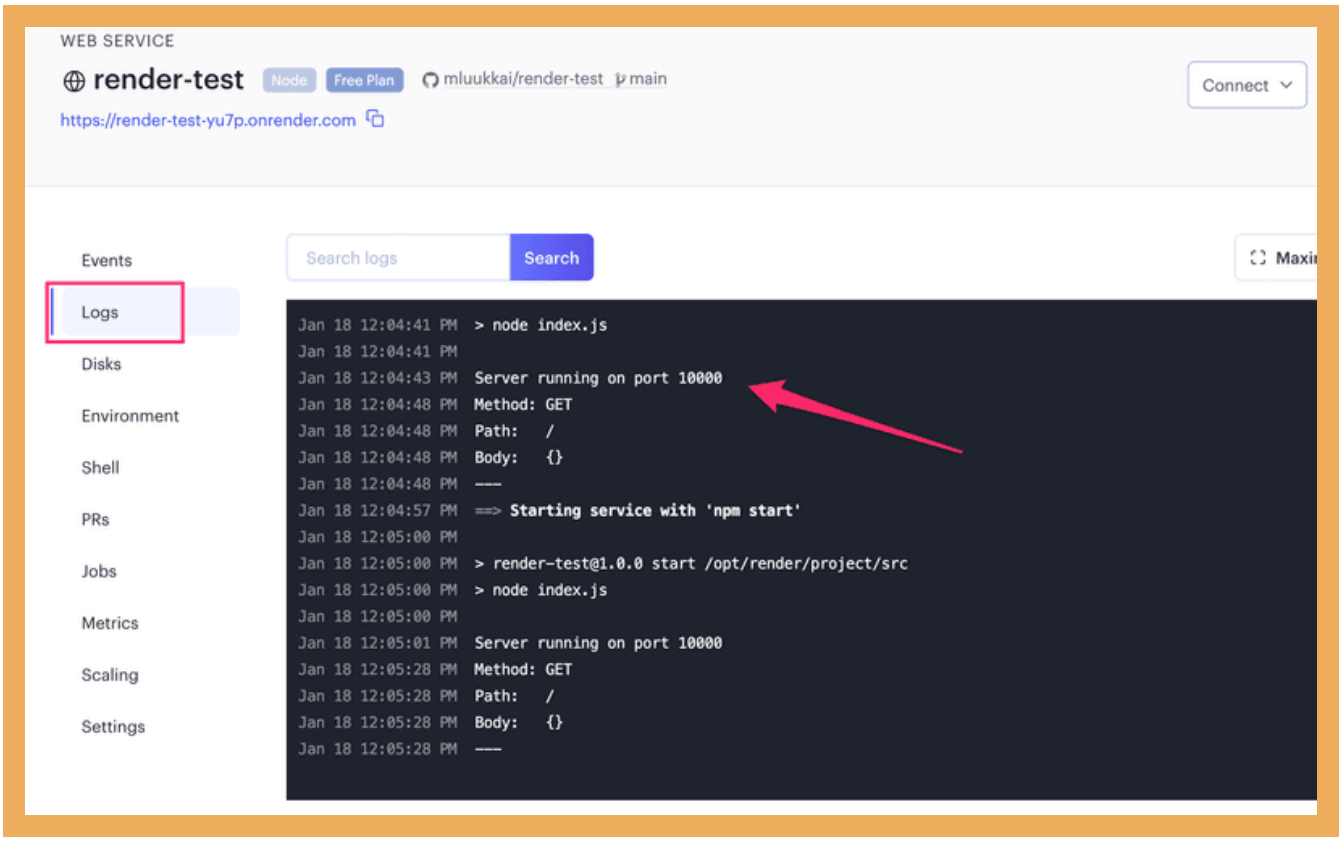
You will also need to whitelist the the fly.io app's IP address in MongoDB Atlas. If you don't MongoDB will refuse the connection.

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.

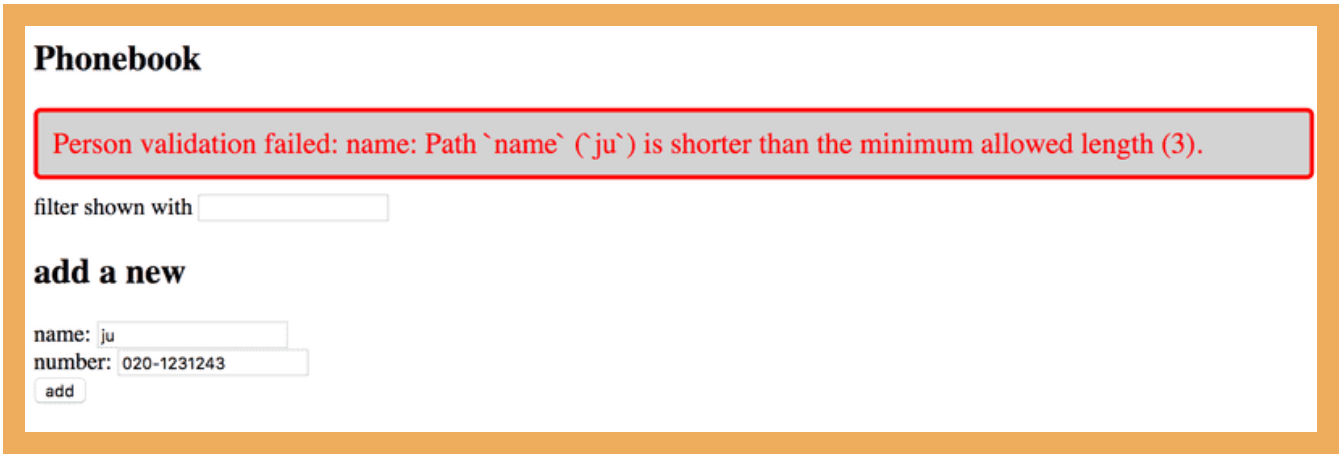
3.19*: Phonebook database, step 7

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 --save-dev
```

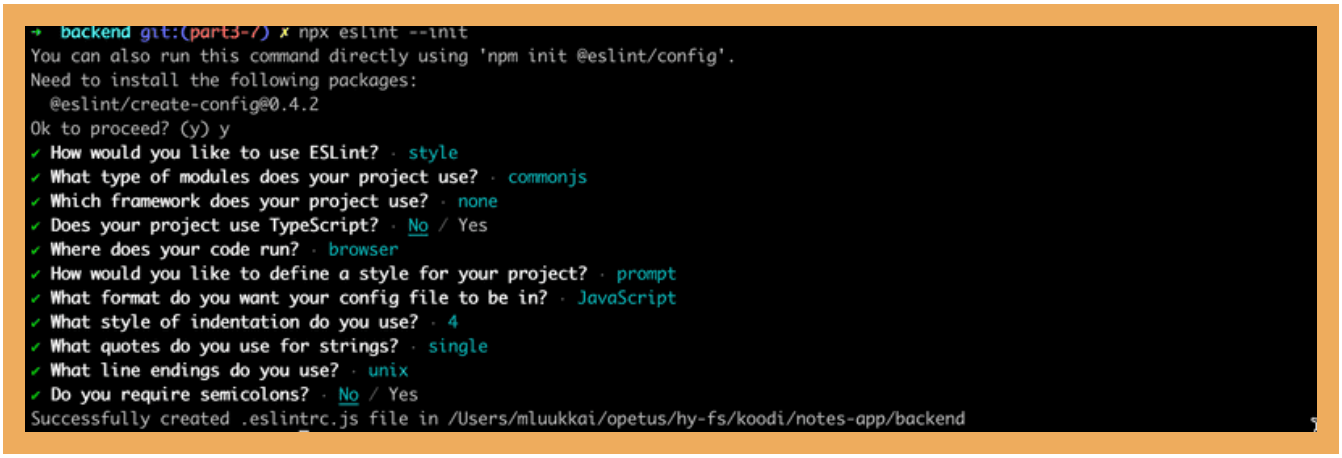
copy

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

```
npx eslint --init
```

copy

We will answer all of the questions:



The configuration will be saved in the `.eslintrc.js` file. We will change `browser` to `node` in the `env` configuration:

```
module.exports = {
    "env": {
        "commonjs": true,
        "es2021": true,
        "node": true
    },
    "overrides": [
        {
            "env": {
                "node": true
            },
            "files": [
                ".eslintrc.{js,cjs}"
            ],
            "parserOptions": {
                "sourceType": "script"
            }
        }
    ],
    "parserOptions": {
        "ecmaVersion": "latest"
    },
    "rules": {
    }
}
```

copy

Let's change the configuration a bit. Install a plugin that defines a set of code style-related rules:

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

Enable the plugin and add an "extends" definition and four code style rules:

```
module.exports = {
    // ...
    'plugins': [
        '@stylistic/js'
    ],
    'extends': 'eslint:recommended',
    'rules': {
        '@stylistic/js/indent': [
            'error',
            2
        ],
        '@stylistic/js/linebreak-style': [
            'error',
            'unix'
        ],
        '@stylistic/js/quotes': [
            'error',
            'single'
        ],
        '@stylistic/js/semi': [
            'error',
            'never'
        ],
    }
}
```

Extends `eslint:recommended` adds a set of recommended rules to the project. In addition, rules for indentation, line breaks, hyphens 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.

Also the files in the `dist` 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:

```
dist
```

copy

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

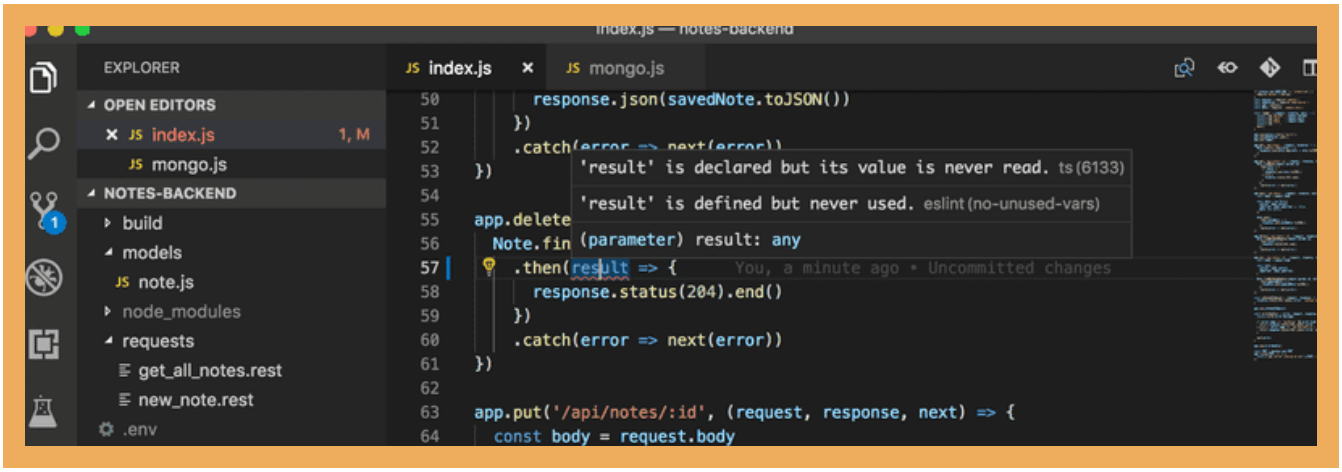
Lint has quite a lot to say about our code:

```
> 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) ✗
```

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 *.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',
```

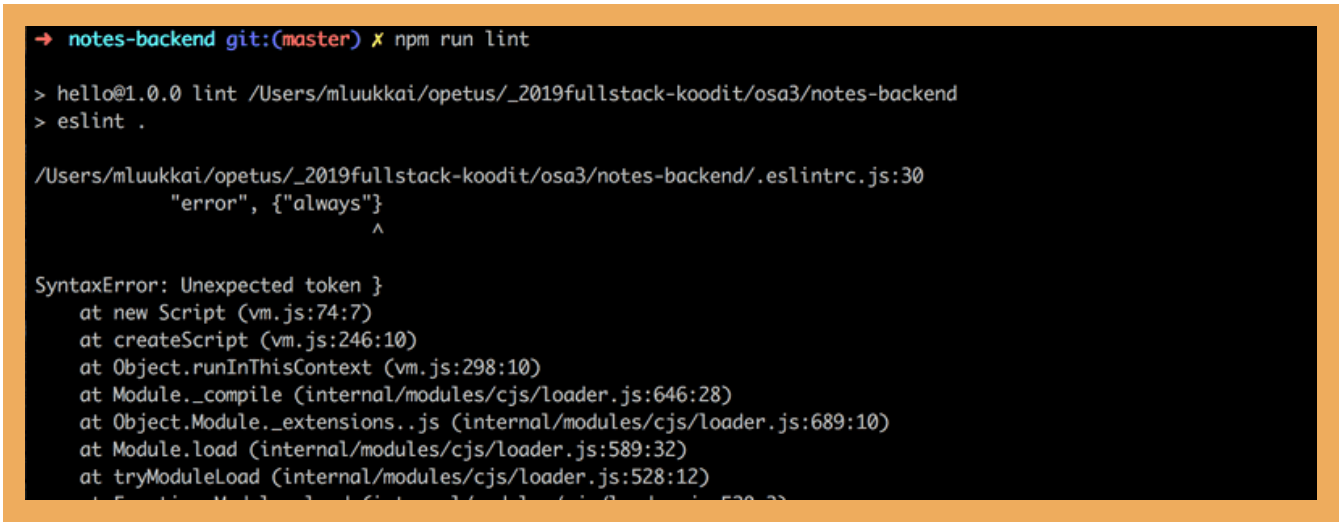
copy

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
  },
}
```

copy

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:



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.

Propose changes to material

Part 3c
Previous part

Part 4
Next part

About course

Course contents

FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON